

NECパーソナルコンピュータ
PC-9800シリーズ

NEC

Software Library

MS-DOS™3.3C
プログラマーズリファレンス
マニュアル Vol.1





Software Library

MS-DOS™3.3C
プログラマーズリファレンス
マニュアル Vol.1

ご注意

- (1) 本書の内容の一部又は全部を無断転載することは禁止されています。
- (2) 本書の内容に関しては将来予告なしに変更することがあります。
- (3) 本書は内容について万全を期して作成いたしましたが、万一御不審な点や誤り、記載もれなどお気づきのことがありましたら御連絡下さい。
- (4) 運用した結果の影響について(3)項にかかわらず責任を負いかねますので御了承下さい。

Microsoft (マイクロソフト) のロゴは米国マイクロソフト社の商標です。

MS-DOS は米国マイクロソフト社の商標です。

CP/M は米国デジタルリサーチ社の登録商標です。

Intel (インテル) は米国インテル社の商標です。

Original Copyright © 1981, 1983, 1984 Microsoft Corporation

Copyright © 1985, 1988, 1989, 1990 NEC Corporation

Translation © 1983, 1985, 1989, 1990 ASCII Techwrite/NEC Corporation

輸出する際の注意事項

本製品 (ソフトウェア) は、外国為替および外国貿易管理法の規定により、戦略物資等輸出規制品に該当します。従って、日本国外に持出す際には日本国政府の輸出許可申請等必要な手続きをお取り下さい。

はじめに

MS-DOS プログラマーズリファレンスマニュアル(Vol. 1/Vol. 2)は、システムプログラマーの方のために、MS-DOS のもとで動作するプログラムを開発する際に必要な、MS-DOS の技術情報を提供するものです。

なお、このマニュアルを十分に利用していただくためには、ソフトウェアおよびハードウェアに関してある程度の専門的な知識が必要となります。

Vol. 1, Vol. 2 のそれぞれで、次のような内容が扱われています。

Vol. 1 標準的な MS-DOS に関して、次のような内容が扱われています。

- ・割り込みとシステムコールの使い方、および用例
- ・デバイスドライバに関する解説、およびデバイスドライバの実例
- ・その他の MS-DOS 技術資料

Vol. 2 標準の MS-DOS から拡張された機能に関して、次のような内容が扱われています。

- ・本体の拡張機能呼び出しに関する解説
- ・特殊なデバイスドライバに関する解説
- ・EMS インターフェイスに関する解説

□本書(Vol. 1)の構成

本書は1章から7章で構成されています。

第1章 MS-DOS で使用できる割り込みとシステムコールを、用例とともに解説しています。

第2章 デバイスドライバについて解説します。デバイスドライバの作成、および MS-DOS へのインストール(登録)のために必要な情報を、具体的なデバイスドライバの例とともに解説しています。

第3章から第7章は、MS-DOS の技術資料です。

第3章 ディスクアロケーションに関する技術資料です。

第4章 コントロールブロックとワークエリアに関する技術資料です。

第5章 EXE 形式のファイルの構造とローディングに関する技術資料です。

第6章 オブジェクトファイルの形式に関する技術資料です。

第7章 プログラム作成に役立つヒントが解説されています。

目 次

第 1 章 システムコール

1.1	イントロダクション	1
1.2	標準キャラクタデバイス I/O	2
1.3	メモリ管理	3
1.4	プロセス管理	5
1.4.1	プログラムのロードと実行	6
1.4.2	オーバーレイのロード	7
1.5	ファイル・ディレクトリ管理	8
1.5.1	ハンドル	8
1.5.2	ファイル管理のファンクションリクエスト	8
1.5.3	デバイス管理のファンクションリクエスト	10
1.5.4	ディレクトリ管理のファンクションリクエスト	11
1.5.5	ディレクトリエントリ	12
1.5.6	ファイルアトリビュート(属性)	12
1.6	MS-Networks	13
1.7	その他のシステム管理	14
1.8	V2.0 以前のシステムコール	15
1.8.1	ファイルコントロールブロック	16
1.8.2	FCB のフィールド	17
1.8.3	拡張 FCB	18
1.9	システムコールの使い方	19
1.9.1	割り込みの使い方	19
1.9.2	ファンクションリクエストの使い方	19
1.9.3	高級言語からのコール	20
1.9.4	レジスタの処理	20
1.9.5	エラー処理	20
1.9.6	システムコールの解説方法	24
1.10	割り込み	26

Program Terminate	27
Function Request	29
Terminate Address	30
CTRL-C Exit Address	30
Critical Error Handler Address	30
1.10.1 エントリの状態	31
1.10.2 割り込みタイプ 24 H ハンドラの必要条件	31
Absolute Disk Read	36
Absolute Disk Write	38
Terminate But Stay Resident	41
1.11 ファンクションリクエスト	43
Terminate Program	49
Display Character	52
Auxiliary Output	54
Direct Console I/O	58
Read Keyboard	62
Buffered Keyboard Input	66
Flush Buffer, Read Keyboard	70
Reset Disk	72
Open File	75
Search For First Entry	80
Delete File	85
Sequential Write	89
Rename File	93
Set Disk Transfer Address	97
Get Drive Data	101
Random Write	106
Set Relative Record	112
Create New PSP	117
Random Block Write	121
Get Date	127
Get Time	131
Set/Reset Verify Flag	135
Get MS-DOS Version Number	139
Keep Process	141
Read Keyboard And Echo	51
Auxiliary Input	53
Print Character	56
Direct Console Input	60
Display String	64
Check Keyboard Status	68
Select Disk	73
Close File	78
Search For Next Entry	83
Sequential Read	87
Create File	91
Get Current Disk	95
Get Default Drive Data	99
Random Read	104
Get File Size	109
Set Interrupt Vector	115
Random Block Read	118
Parse File Name	124
Set Date	129
Set Time	133
Get Disk Transfer Address	137
CTRL-C Check	143

Get Interrupt Vector	145	Get Disk Free Space	147
Get Country Data	149	Set Country Data	153
Create Directory	155	Remove Directory	157
Change Current Directory	159		
Create Handle	161	Open Handle	163
Close Handle	167	Read Handle	169
Write Handle	171	Delete Directory Entry	173
Move File Pointer	175	Get/Set File Attributes	178
Get IOCTL Data	181	Set IOCTL Data	184
Receive IOCTL Character	186	Send IOCTL Character	188
Receive IOCTL Block	190	Send IOCTL Block	192
Get Input IOCTL Status	194	Get Output IOCTL Status	196
IOCTL Is Changeable	198	IOCTL Is Redirected Block	200
IOCTL Is Redirected Handle	202		
IOCTL Retry	204	Generic IOCTL (for handles)	206
Generic IOCTL (for block devices)	207		
Get/Set Logical Drive Map	213		
Duplicate File Handle	214	Force Duplicate File Handle	216
Get Current Directory	218	Allocate Memory	220
Free Allocated Memory	223	Set Block	226
Load and Execute Program	228		
Load Overlay	232	End Process	235
Get Return Code Child Process	237		
Find First File	238	Find Next File	240
Get Verify State	242	Change Directory Entry	244
Get/Set Date/Time of File	246		
Get/Set Allocation Strategy	249		
Get Extended Error	252	Create Temporary File	255
Create New File	258	Lock	260
Unlock	264	Get Machine Name	266
Printer Setup	268	Get Assign List Entry	270
Make Assign List Entry	273		
Cancel Assign List Entry	276		
Get PSP	278		

1.12 MS-DOS システムコールにおけるマクロ定義例	279
1.13 MS-DOS システムコールにおける拡張例	295

第2章 MS-DOS デバイスドライバ

2.1 デバイスドライバとは	301
2.2 デバイスドライバのフォーマット	302
2.3 デバイスドライバの作成方法	304
2.3.1 デバイスストラテジルーチン	305
2.3.2 デバイス割り込みルーチン	305
2.4 デバイスドライバの登録	305
2.5 デバイスヘッダ	306
2.5.1 つぎのデバイスヘッダフィールドに対するポインタ	307
2.5.2 アトリビュート(属性)フィールド	307
2.5.3 ストラテジと割り込みルーチン	308
2.5.4 名前フィールド	309
2.6 リクエストヘッダ	309
2.6.1 レコード長	309
2.6.2 ユニットコード	310
2.6.3 コマンドコードフィールド	310
2.6.4 ステータスフィールド	311
2.7 デバイスドライバファンクション	312
2.7.1 INIT	312
2.7.2 MEDIA CHECK	315
2.7.3 BUILD BPB	317
2.7.4 リードまたはライト	319
2.7.5 連続非破壊読み込み	321
2.7.6 オープンまたはクローズ	321
2.7.7 REMOVABLE MEDIA	322
2.7.8 STATUS	323
2.7.9 FLUSH	323
2.7.10 Generic IOCTL	324
2.7.11 DEINSTALL	325
2.7.12 Get/Set Logical Drive Map	325
2.8 メディアディスクリプタバイト	326
2.9 メディアディスクリプタテーブル	326

2.10	クロックデバイス	329
2.11	デバイスコールの分析	329
2.12	デバイスドライバ例	331
2.12.1	ブロックデバイスドライバ	331
2.12.2	キャラクタデバイスドライバ	337

第3章 MS-DOS 技術資料

3.1	MS-DOS の初期化	345
3.2	コマンドプロセッサ	345
3.3	MS-DOS ディスクアロケーション	346
3.4	MS-DOS ディスクディレクトリ	346
3.5	MS-DOS ファイルアロケーションテーブル	349
3.5.1	12 ビット FAT エントリ	350
3.5.2	16 ビット FAT エントリ	351

第4章 MS-DOS コントロールブロックとワークエリア

4.1	MS-DOS メモリマップ	353
4.2	MS-DOS プログラムセグメント	354

第5章 EXE ファイルの構造とローディング

EXE ファイルの構造とローディング	359
--------------------	-----

第6章 インテルオブジェクトモジュールフォーマット

6.1	イントロダクション	363
6.2	用語の定義	364
6.3	モジュールの一致と属性	367
6.4	セグメント定義	367
6.5	セグメントアドレッシング	368
6.6	シンボル定義	368
6.7	インデックス	369
6.8	フィックスアップのためのフレームの概念	369

6.9	セルフリラティブフィックスアップ	374
6.10	セグメントリラティブフィックスアップ	375
6.11	レコードオーダ	376
6.12	レコードフォーマットについて	377
6.12.1	SAMPLE RECORD FORMAT	377
6.12.2	T-MODULE HEADER RECORD	379
6.12.3	LIST OF NAMES RECORD	379
6.12.4	SEGMENT DEFINITION RECORD	380
6.12.5	GROUP DEFINITION RECORD	383
6.12.6	TYPE DEFINITION RECORD	384
6.12.7	PUBLIC NAMES DEFINITION RECORD	387
6.12.8	EXTERNAL NAMES DEFINITION RECORD	389
6.12.9	LINE NUMBER RECORD	390
6.12.10	LOGICAL ENUMERATED DATA RECORD	391
6.12.11	LOGICAL ITERATED DATA RECORD	392
6.12.12	FIXUP RECORD	394
6.12.13	MODULE END RECORD	398
6.12.14	COMMENT RECORD	400
6.13	レコードの番号によるリスト	402
6.14	共有変数の型に関するマイクロソフト表現法	403

第7章 プログラムヒント

7.1	イントロダクション	407
7.2	割り込みタイプ	407
7.3	システムコール(ファンクションリクエスト)	408
7.4	デバイス管理	409
7.5	メモリ管理	409
7.6	プロセス管理	410
7.7	ファイル・ディレクトリ管理	410
7.8	その他のプログラム手順	411
索引	413

第1章

システムコール

1.1 イントロダクション

MS-DOS ではシステムを操作・管理するためのサブルーチン（システムコール）をアプリケーションプログラムからコールすることができます。

システムコールを利用すれば、機種に依存しないプログラムを簡単に作成することができ、また将来の MS-DOS のバージョンに対しても問題なく動作する可能性が高くなります。

本書では、MS-DOS システムコールをつぎのように分類して解説を行います。

- 標準キャラクタデバイス I/O
- メモリ管理
- プロセス管理
- ファイル・ディレクトリ管理
- MS-Networks コール
- その他のシステムファンクション

MS-DOS システムコールは、ソフトウェア割り込みによってアプリケーションプログラムから利用できます。

通常、MS-DOS で使用する割り込みタイプは、20 H～27 H と予約されている 28 H～3FH です。

割り込みタイプ 21H は特に“ファンクションリクエスト”と呼ばれ、MS-DOS のサポートする大半の機能を利用することができます。

割り込みタイプ 21H の選択はアプリケーションプログラムがファンクション番号を AH レジスタ（いくつかのファンクションリクエストでは、AX レジスタ全部）にセットすることで行われます。

各ソフトウェア割り込み、ファンクションリクエストは、ファンクション指定情報の受け渡しのためにレジスタを用います。

変更されたシステムコール

V 2.0 (バージョン 2.0) 以前の MS-DOS で紹介された多くのシステムコールよりも、簡単にシステム資源を利用できるように、V 2.0 以降では、その代用となるファンクションリクエストを用意しています。今後の MS-DOS のバージョンで付加される機能に対応するために、新しいファンクションコールの使用をお奨めします。

しかし、V 2.0 以前のシステムコールを使っているプログラムを単に V 2.0 以降の MS-DOS に対応するために書き換える必要はありません。MS-DOS は互換性を保つために V 2.0 以前のシステムコールもサポートしています。

V 2.0 以前のシステムコールの一覧とファイルコントロールブロック (FCB) の解説については、1.8 “V 2.0 以前のシステムコール” を参照してください。

本書第1章の前半は、メモリ、ファイル、プロセスなどに関する MS-DOS のシステム管理の方法と、システムコールの大半を目的別に解説しています。後半では、各ソフトウェア割り込みとファンクションリクエストの詳細を個別に解説しています。システムコールの解説はソフトウェア割り込みとファンクションリクエストです。これらの解説には MS-DOS のシステム管理についての詳細な説明も含まれています。

第2章はデバイスドライバの作成方法です。第3章から第5章は MS-DOS の詳細な情報で、MS-DOS ディスクアロケーション、MS-DOS コントロールブロックとワークエリア作業領域、EXE 形式のファイルの構造とローディングが解説されています。第6章ではインテルオブジェクトモデルの構造について、第7章ではプログラムを作る上でのヒントについて、それぞれ述べられています。

1.2 標準キャラクタデバイス I/O

標準キャラクタのファンクションリクエストを用いると、コンソール、プリンタ、シリアルポートなどのキャラクタデバイスをすべて同じように、入出力操作することができます。

つぎの表は標準キャラクタデバイス入出力の MS-DOS ファンクションリクエストの一覧です。プログラムでこれらのファンクションリクエストを用いると、入出力のリダイレクトができます。

表 1.1 標準キャラクタデバイス入出力のファンクションリクエスト

コード	機 能	説 明
01H	キーボード入力とエコー	標準入力から 1 文字を受け取ると、その文字を標準出力に出力します。
02H	文字のスクリーン出力	標準出力に 1 文字を出力します。
03H	補助入力	補助入力装置から 1 文字を受け取ります。
04H	補助出力	補助出力装置に 1 文字を出力します。
05H	文字のプリンタ出力	プリンタに 1 文字を出力します。
06H	直接コンソール I/O	標準入力から 1 文字を受け取るか、標準出力に 1 文字を出力します。
07H	直接コンソール入力	標準入力から 1 文字を受け取ります。
08H	キーボード入力	標準入力から 1 文字を受け取ります。
09H	ストリングのスクリーン出力	標準出力に文字列を出力します。
0AH	バッファードキーボード入力	標準入力から文字列を受け取ります。
0BH	キーボードステータスの検査	標準入力のバッファのステータスを返します。
0CH	バッファを空にして、キーボード入力	標準入力のバッファを空にして標準入力から 1 文字を受け取ります。

標準キャラクタ I/O ファンクションリクエストのいくつかは同じ機能を持っていますが、キャラクタを標準入力から標準出力にエコーするか、コントロールキャラクタをチェックするかどうかで区別されます。この違いについての詳細は、1.9～1.11 を参照してください。

1.3 メモリ管理

MS-DOS は、各メモリ領域の先頭のメモリコントロールブロックによってメモリ領域の割り当てを管理しています。このメモリコントロールブロックには、そのメモリ領域が使われているか、使用中ならばそのメモリブロックを要求したプロセスの PSP のセグメントアドレス、このメモリコントロールブロックが管理するメモリブロックのサイズなどが書き込まれています。もし、あるメモリ領域が使われていなければ、他のプロセスで利用することができます。

つぎの表はメモリ管理の MS-DOS ファンクションリクエストの一覧です。

表 1.2 メモリ管理ファンクションリクエスト

コード	機 能	説 明
48H	メモリの割り当て	メモリの割り当てを要求する
49H	割り当てられたメモリの解放	割り当てられたメモリを解放します。
4AH	割り当てられたメモリブロックの変更	割り当てられたメモリブロックを変更します。

プロセスがファンクション 48H によって、メモリの割り当てを要求すると、MS-DOS は要求を満たす大きさの空きメモリブロックを捜します。条件に見合う空きメモリブロックが見つかり、MS-DOS はそのメモリコントロールブロックを修正し、要求を出したプロセスの所有するメモリとします。

もし、空きメモリブロックが要求量よりも大きかったときは、まずメモリコントロールブロックのサイズフィールドを要求量に合うように修正し、必要量をプロセスに割り当てます。つぎに、残った空きメモリ領域の先頭に新しいメモリコントロールブロックを作成し、ポインタを更新して、このメモリブロックをメモリコントロールブロックのチェーン（連鎖）に加えます。そして、MS-DOS はメモリを要求したプロセスへ、割り当てたメモリブロックの先頭バイトのセグメントアドレスを返します。

プロセスがファンクション 49H によってメモリブロックを解放したときは、MS-DOS はメモリコントロールブロックを利用可能（いずれのプロセスにも所有されていない）なものとなります。

プロセスがファンクション 4AH を使って、メモリブロックサイズを縮小させたときは、MS-DOS は縮小によって解放されたメモリ領域の先頭にメモリコントロールブロックを作成し、メモリコントロールブロックのチェーンに加えます。

プロセスがファンクション 4AH を使って、メモリブロックサイズを拡大させたときは、MS-DOS はメモリブロックを割り当てるときと同様に扱いますが、このときはセグメントアドレスは返さず、たんに追加メモリブロックとそれまでのメモリブロックをチェーンします。

もし、ファンクション 48H または 4AH で、要求量を満たす空きメモリブロックが見つからなかった場合には、MS-DOS はメモリを要求したプロセスにエラーコードを返します。

プログラム（プロセス）は制御が渡されたときに、まずファンクション 4AH によって、メモリアロケーションブロック（PSP から始まる）の初期設定値を修正し、ブロックを必要なだけ

の大きさに切り詰めるとよいでしょう。この処置は不必要なメモリ領域を解放します。また、それによってプログラムは将来のマルチタスク処理に対して移植性の高いものになります。

プログラムがEXIT (終了) するときは、呼び出したプログラム (アプリケーションプログラムを呼び出すのは通常 COMMAND.COM です) がコントロールを取り戻す前に、MS-DOS は自動的にメモリアロケーションブロックを解放します。プロセスがEXIT することにより、MS-DOS は占有されていたメモリをすべて解放します。

どのようなプログラムも、メモリコントロールブロックをこわしてはなりません。もしメモリコントロールブロックのチェーンがこわれると、メモリアロケーションエラーとなり、システムを再起動しなければなりません。

1.4 プロセス管理

MS-DOS はプログラムのロード、実行、終了などのプロセスに関する種々のファンクションコールを備えています。アプリケーションプログラムでも、これらのファンクションコールを使って他のプログラムの管理ができます。

つぎの表は、プロセス管理のための MS-DOS ファンクションリクエストの一覧です。

表1.3 プロセス管理ファンクションリクエスト

コード	機 能	説 明
31H	キーププロセス	プログラムを終了させ、呼び出したプログラムに制御を返しますが、終了したプログラムはメモリ上に残っています。
4B00H	プログラムのロードと実行	プログラムをロードし、実行します。
4B03H	オーバーレイのロード	プログラムをロードしますが、実行しません。
4CH	プロセスの終了	呼び出したプログラムに制御を返します。
4DH	子プロセスからリターンコードを得る	子プロセスがEXIT したときのコードを返します。
62H	PSP コードを得る	カレントプロセスのプログラムセグメントの先頭のセグメントアドレスを返します。

1.4.1 プログラムのロードと実行

ファンクション 4B00H によって、あるプログラムが別のプログラムを起動するとき、つぎのような手順の処理が行われます。

まず、MS-DOS によって、メモリが割り当てられます。つぎに、割り当てられたメモリの先頭（オフセット 0）に、新しいプログラムセグメントプレフィクス（PSP）が書き込まれます。続いて、プログラムがロードされ、プロセスの制御が目的のプログラムに渡されます。呼び出されたプログラムが EXIT（終了）したときには、制御は呼び出したプログラムに戻ります。

COMMAND.COM は、ファンクション 4B00H を使ってコマンドをロードし、実行しています。アプリケーションプログラムで、COMMAND.COM 以上のプロセス管理をすることもできます。

MS-DOS で実行可能なプログラムには、COM 形式（.COM の拡張子を持つ）と EXE 形式（.EXE の拡張子を持つ）の 2 種類の形式があります。いままでに扱った事柄は、両形式に共通の部分でした。以下では、両者の違いを述べます。

COM 形式のロードと実行

COMMAND.COM は、COM 形式プログラムをロードし、実行するとき、空メモリ領域すべてをアプリケーションに割り当て、プログラムに 64 KB 以上のメモリを割り当てることができるならば、オフセット 0000 H を SP にセットし、スタックに 0 を PUSH して SP=FFFEH とします。割り当てるメモリが 64 KB よりも少ないときプログラムの最上位オフセット+1 を SP にセットし、0 を PUSH します。

COM 形式のプログラムは、ファンクション 4AH でメモリアロケーションブロックの初期値を縮小される前に、スタック領域を確保します。なぜなら、既定のスタック領域は解放されるメモリ領域にあるからです。

もし、新たにロードされたプログラムが、COM 形式プログラムのように、すべてのメモリを割り当てられたり、ファンクション 48 H によって空領域のすべてを要求すると、MS-DOS は COMMAND.COM の非常駐部分も割り当てます。

プログラムがこの領域を変化させて終了したときは、MS-DOS は COMMAND.COM の非常駐部を再ロードしてから、制御を COMMAND.COM に戻します。

もし、プログラムがメモリを十分に解放せずに（ファンクションリクエスト 31 H を使って）常駐終了した場合、COMMAND.COM の非常駐部を再ロードできずシステムが停止する恐れがあります。COM 形式のプログラムは、このような事態の発生を最小限に抑えるために、事前にファンクション 4AH を使って、分割されるブロックの初期値を小さくし、プログラムが常駐終了する前に、ファンクション 49 H ですべてのメモリを解放するようにします。

EXE 形式のロード方法

COMMAND.COM が、EXE 形式のプログラムをロード、実行するときは、つぎのような手順

で行われます。

まず、EXE 形式のプログラム自体のサイズ (メモリイメージで) に、そのプログラムが必要とするメモリ量を確保します。このメモリ量は、メモリが十分にある場合は、ファイルヘッダの MAXALLOC フィールド (オフセット 0CH) の値、不十分な場合は MINALLOC フィールド (オフセット 0AH) の値です。なお、これらのフィールドの値は、リンカによって設定されます。

つぎに、MS-DOS はファイルヘッダの情報によって、EXE 形式のファイルの実アドレスを決定して、プログラムをロードします。

その後、制御がプログラムに引き渡されます。

MS-DOS における、COM 形式と EXE 形式のプログラムのロードの詳細については、第 3 章、第 4 章をご覧ください。

プログラムから別のプログラムを実行する方法

COMMAND.COM はパスを設定したり、パスを使って実行可能なプログラムを捜したり、EXE 形式のファイルをリロケート (再配置) するなどの細かい処理まで行います。したがって、常駐とは別に COMMAND.COM をコピーし、その COMMAND.COM の実行を通して別のプログラムをロード、実行することによって、あるプログラムから別のプログラムを実行する方法が最も簡単です。そのためには、コマンドラインに /C スイッチをつけて、目的のプログラムを起動します (詳しくはファンクション 4B00H の解説を参照してください)。

1.4.2 オーバーレイのロード

ファンクション 4B03H を使って、オーバーレイ形式のプログラムをロードするときに、プログラムはオーバーレイ部分がロードされるセグメントアドレスを MS-DOS に知らせなければなりません。プログラムはオーバーレイをコールするときに、セグメントアドレスを知らせ、オーバーレイは呼んだプログラムヘディレクトリを返します。オーバーレイを呼んだプログラムは、それを完全にコントロールする必要があります。MS-DOS はオーバーレイに対して PSP を書き込んだり、他の方法で干渉することはありません。

MS-DOS は、呼んだプログラムの持っている (使っている) メモリ領域にオーバーレイがロードされた場合、その事をチェックしません。

もし、呼んだプログラムが十分なメモリを持たずにオーバーレイをロードすると、メモリコントロールブロックがこわれ、致命的なメモリアロケーションエラーが生じます。

それゆえに、オーバーレイをロードするプログラムは、ファンクション 4AH を使ってメモリアロケーションブロックの初期値を縮小するときに、必ずオーバーレイを格納する場所を用意するか、メモリアロケーションブロックの初期値を最小に縮小してから、ファンクション 48H を使ってオーバーレイのためにメモリを割り当てます。

1.5 ファイル・ディレクトリ管理

MS-DOS は、ファイルやディレクトリを管理するためのさまざまなファンクションコールを備えています。

1.5.1 ハンドル

ファイルを作成したり、オープンしたりするために、プログラムは MS-DOS にパス名やファイルを割り付けるための属性を渡します。MS-DOS はハンドルと呼ばれる 16ビットの数字を返します。以後、MS-DOS は続く操作のためにこのハンドルと一致するファイルだけを捜します。

ハンドルはファイルかデバイスのどちらかと対応することができます。MS-DOS では5つの標準ハンドルをあらかじめ設定しています。これらは常にオープンされているので、使用の際にあらためてオープンする必要はありません。つぎの表はその一覧です。

表 1.4 デバイスハンドル

ハンドル	標準デバイス名	備考
0	標準入力	リダイレクト可
1	標準出力	リダイレクト可
2	エラー出力	
3	補助装置	
4	プリンタ	

MS-DOS はファイルを作成したりオープンしたりするときに、利用可能な最初のハンドルを割り付けます。1つのプログラム(またはプロセス)がオープンできるハンドルの数は20で、この中には先の5つの標準デバイスが含まれますから普通1つのプログラムでオープンできるファイルの数は15です。5つの標準デバイスのいずれも、ファンクション 46 H を使って、ファイルやデバイスを一時的に結び付けることができます。

1.5.2 ファイル管理のファンクションリクエスト

MS-DOS は、ファイルを単純なバイト列として扱います。したがって、レコード構造体や、それに関する特別なアクセス技術はありません。アプリケーションプログラムでは、構造体でも、バイト列として扱います。ファイルの読み出し／書き込み処理は、データバッファへのポインタと、読み書きするバイト数だけを必要とします。

つぎの表はファイル管理のファンクションリクエストです。

表 1.5 ファイル管理のファンクションリクエスト

コード	機 能	説 明
3CH	ハンドルの作成	ファイルの作成
3DH	ハンドルのオープン	ファイルのオープン
3EH	ハンドルのクローズ	ファイルのクローズ
3FH	リードハンドル	ファイルの読み出し
40H	ライトハンドル	ファイルの書き込み
42H	ファイルポインタの移動	読み書きするファイル中のポインタの移動
45H	ファイルハンドルの二重化	すでにオープンされている他のハンドルを同じハンドルとして扱います。新規のハンドルを返します。
46H	ファイルハンドルの強制二重化	すでにオープンされているハンドルと、すでにオープンされている他のハンドルとを、同じファイルとして扱うよう強制します。
5AH	一時ファイルの作成	ユニークな名前のファイルを作成します。
5BH	新しいファイルの作成	新しいファイルを作成します。しかし、同じファイル名が存在する場合はファイルを作成しません。

ファイルシェアリング

MS-DOS 3.3 では、1つ以上のプロセスがファイルを共有してアクセスするファイルシェアリングを導入しています。ファイルシェアリングは、ファイルシェアリングをサポートするためのシェアリングコマンドが実行された後にだけ動作します。つぎの表はファイルシェアリングに使われるファンクションリクエストの一覧です。

表 1.6 ファイルシェアリング ファンクションリクエスト

コード	機 能	説 明
3DH	ハンドルのオープン	ファイルシェアリングモードにして、1つのファイルをオープンします。
440BH	IOCTL リトライ	ファイルシェアリングの暴走による I/O 操作の失敗に対して、割り込みタイプ 24H を実行する前に再試行する回数を設定します。
5C00H	ファイルアクセスのロック	ファイルの一部をロックします。
5C01H	ファイルアクセスのロック解除	ファイルの一部のロックを解除します。

ファイルシェアリングが有効でない場合、これらのファンクションリクエストは使用できません。ファンクション 3DH (ハンドルのオープン) は種々のモードで動作します。コンパチビリティモードはファイルシェアリングが有効でなくても使えます。ファイルシェアリングモードは、ファイルシェアリングが有効な (事前に SHARE.EXE が実行され、メモリに常駐している) ときに意味を持ちます。

1.5.3 デバイス管理のファンクションリクエスト

異なるデバイスを扱う種々のコードを含むファンクション 44H を使って、デバイスに対して I/O コントロールを実行します。このファンクションは異なるデバイスを扱う種々のコードを含みます。IOCTL ファンクションリクエストのいくつかは、デバイスドライバが IOCTL ファンクションをサポートするのに使用されます。つぎの表は MS-DOS のデバイス管理のファンクションリクエストの一覧です。

表 1.7 デバイス管理ファンクションリクエスト

コード	機 能	説 明
4400H	IOCTL データを得る。	デバイスの種類の取得
4401H	IOCTL データをセットする。	デバイスの種類のセット
4402H	IOCTL キャラクタを受け取る。	キャラクタデバイスのコントロールデータを受け取る。
4403H	IOCTL キャラクタを送る。	キャラクタデバイスのコントロールデータを送る。
4404H	IOCTL ブロックを受け取る。	ブロックデバイスのコントロールデータを受け取る。
4405H	IOCTL ブロックを送る。	ブロックデバイスのコントロールデータを送る。
4406H	入力ステータスのチェック	デバイスのステータスが入力かどうかをチェックします。
4407H	出力ステータスのチェック	デバイスのステータスが出力かどうかをチェックします。
4408H	IOCTL の交換性	ブロックデバイスが差し換え可能な媒体を含んでいるかをチェックします。
440CH	一般 IOCTL (ハンドル用)	プリンタに対して、出力繰り返し回数の設定と取得をします。
440DH	一般 IOCTL (ブロックデバイス用)	論理デバイスに対して、リード、ライト、フォーマット、ベリファイを行います。
440EH	論理ドライブマップの取得	現在の、論理ドライブと物理ドライブのマップ情報を取得します。
440FH	論理ドライブマップの設定	論理ドライブを物理デバイスにマップします。

IOCTL ファンクションリクエストのいくつかの形式は、MS-Networks でしか使えません。詳しくは “1.6 MS-Networks” を参照してください。

1.5.4 ディレクトリ管理のファンクションリクエスト

ディスクのルートディレクトリが管理できるディレクトリ、ファイル名の数にはメディアに依存する上限があります。ハードディスクでは、ディレクトリ、ファイル名数はMS-DOSのパーティション容量に依存します。サブディレクトリは特別な属性を持ったファイルであり、サブディレクトリ下に作成できるディレクトリ、ファイル数はディスクの空容量にのみ制限されます。また、パス名の長さは64文字（英数字）を超えることはできません。

ルートディレクトリに関しては、V 2.0 以前とまったく変わっていません。また、V 2.0 以前のMS-DOSで作成されたディスクは、サブディレクトリをサポートしていないので、単にルートディレクトリだけを持つものとして扱われます。サブディレクトリはV 2.0 からサポートされており、V 2.0 以降は互換性があります。

つぎにディレクトリ管理のファンクションリクエストの一覧を示します。

表 1.8 ディレクトリ管理ファンクションリクエスト

コード	機 能	説 明
39H	ディレクトリの作成	サブディレクトリの作成
3AH	ディレクトリの削除	サブディレクトリの削除
3BH	カレントディレクトリの変更	カレントディレクトリの変更
41H	ディレクトリエントリの削除	ファイルの削除
43H	アトリビュート(属性)を得る／セットする	ファイルの属性の取得またはセット
47H	カレントディレクトリのテキストを得る	指定ドライブのカレントディレクトリを返します。
4EH	最初に一致するファイル名の検索	該当するファイル（ワイルドカード等で指定した）を検索します。
4FH	つぎに一致するファイル名の検索	該当するファイル（ワイルドカード等で指定した）の検索を続行します。このファンクションは、ファンクション 4EH に続いて実行されます。
56H	ディレクトリエントリの変更	ファイル名の変更
57H	ファイルの日付／時刻を得る／セットする	ファイルの日付または時刻を取得またはセットします。

1.5.5 ディレクトリエントリ

ディレクトリエントリはファイル名、最後に変更された日付と時刻、ファイルサイズなどを含む 32 バイトのレコードです。サブディレクトリのエントリはルートディレクトリのエントリと同じです。ディレクトリエントリについての詳細は第 3 章を参照してください。

1.5.6 ファイルアトリビュート（属性）

つぎの表は、ファイルアトリビュート（属性）とディレクトリエントリの属性を表すバイト（オフセット 0BH）の一覧です。アトリビュート（属性）は、ファンクション 43H によって調べたり、変えることができます。

表 1.9 ファイルアトリビュート

コード	内 容
00H	普通のファイル。自由に読み出しや書き込みができます。
01H	読み出し専用。書き込むためにファイルをオープンすることも、同じ名前のファイルを作成することもできません。
02H	隠しファイル。DIR コマンドで見ることができません。
04H	システムファイル。DIR コマンドで見ることができません。
08H	ボリューム ID。この属性を持てるファイルは、ルートディレクトリ上に 1 つだけです。
10H	サブディレクトリ。
20H	アーカイブ(保存)ファイル。ファイルが変更されたときに作られ、バックアップコマンドによって消去されます。

ボリューム ID (08H) とディレクトリ (10H) の属性は、ファンクション 43H では、変更できません。

1.6 MS-Networks

MS-Networks は1つ以上のサーバとワークステーションから構成されます。MS-DOS は、サーバに対するワークステーションドライブとワークステーションデバイスの割り当ての情報を保管します。MS-Networks の詳細については、MS-Networks マネージャーズガイドとユーザーズガイドを参照してください。

つぎの表は MS-Networks 管理のファンクションコールの一覧です。

表 1.10 MS-Networks ファンクションリクエスト

コード	機 能	説 明
4409H	IOCTL リダイレクトブロック	ドライブ名によって Networks のワークステーションか、サーバかを調べます。
440AH	IOCTL リダイレクトハンドル	ハンドル名によって Networks のワークステーションか、サーバかを調べます。
5E00H	マシン名を得る	ワークステーションの Networks 名を得ます。
5E02H	プリンタセットアップ	Networks プリンタへ送るファイルの先頭に、コントロールキャラクタをセットします。
5F02H	割り当てリストのエントリを得る	Networks の割り当てリストのエントリ（ワークステーションのドライブ名またはデバイス名、再割り当てされたディレクトリやデバイスの Networks 名など）を得ます。
5F03H	割り当てリストのエントリを作成	ワークステーションのドライブやデバイスから、サーバへリダイレクションを行います。
5F04H	割り当てリストのエントリの取り消し	ワークステーションからサーバへのリダイレクションを取り消します。

1.7 その他のシステム管理

その他のシステムコールは、その他のシステムファンクションとドライブ、クロック、アドレスなどの情報を管理します。

つぎの表は、種々のシステム情報やシステム操作を管理する MS-DOS ファンクションリクエストの一覧です。

表 1.11 その他のシステム管理ファンクションコール

コード	機 能	説 明
0DH	リセットディスク	ファイルバッファを空にします。
0EH	ディスクの選択	デフォルトドライブのセット
19H	カレントディスクを得る	デフォルトのドライブを返します。
1AH	ディスク転送アドレスのセット	ディスク転送バッファのアドレスのセット
1BH	デフォルトのドライブのデータを得る	デフォルトのディスクのフォーマット情報を返します。
1CH	ドライブのデータを得る	ディスクのフォーマット情報を返します。
25H	割り込みベクタのセット	割り込み処理ルーチンのアドレスのセット
29H	ファイル名の解析	ファイル名の解析
2AH	日付を得る	システムの日付を取得します。
2BH	日付のセット	システムの日付をセットします。
2CH	時刻を得る	システムの時刻を取得します。
2DH	時刻のセット	システムの時刻をセットします。
2EH	ベリファイフラグのセット／リセット	ベリファイフラグのセット／リセット
2FH	ディスク転送アドレスを得る	ディスク転送アドレスの取得
30H	MS-DOS バージョン番号を得る	MS-DOS のバージョン番号を返します。
33H	<CTRL-C>検査のセット／リセット	<CTRL-C>検査のステータスを返します。
35H	割り込みベクタを得る	割り込みルーチンのアドレスを返します。
36H	ディスクのフリースペースを得る	ディスクのフリースペースのデータを返します。
38H	国別情報を得る／セットする	国別情報を取得／セットします。
54H	ベリファイの状態を返す	ベリファイのステータスを返します。

1.8 V2.0以前のシステムコール

V2.0以前のシステムコールで、V2.0以降のファンクションリクエストで代用されるものの多くは、ファイルを取り扱うシステムコールです。つぎの表は、これらのV2.0以前のシステムコールとV2.0以降代用されるファンクションリクエストの一覧です。

MS-DOSは現在でも、これらの古いシステムコールを含んでいます。しかし、これはV2.0以前のプログラムの互換性を保つためだけですので、プログラムを作成する際は、V2.0以前のシステムコールを使用しないことをお奨めします。

表 1.12 V2.0以前のシステムコールとその代用となるファンクションリクエスト

V2.0以前のシステムコール	V2.0以降、代用されるファンクションリクエスト
<u>ファンクションリクエスト</u>	<u>ファンクションリクエスト</u>
00H プログラムの終了	4CH プロセスの終了
0FH ファイルのオープン	3DH ハンドルのオープン
10H ファイルのクローズ	3EH ハンドルのクローズ
11H 最初のエントリを検索	4EH 最初に一致するファイル名の検索
12H つぎのエントリを検索	4FH つぎに一致するファイル名の検索
13H ファイルの削除	41H ディレクトリエントリの削除
14H シーケンシャルリード	3FH リードハンドル
15H シーケンシャルライト	40H ライトハンドル
16H ファイルの作成	3CH ハンドルの作成
	5AH 一時ファイルの作成
	5BH 新しいファイルの作成
17H ファイル名の変更	56H ディレクトリエントリの変更
21H ランダムリード	3FH リードハンドル
22H ランダムライト	40H ライトハンドル
23H ファイルの大きさを得る	42H ファイルポインタの移動
24H 相対レコードのセット	42H ファイルポインタの移動
26H 新しいPSPを作成する	4B00H プログラムのロードと実行
27H ランダムブロックリード	3FH リードハンドル
28H ランダムブロックライト	40H ライトハンドル
<u>ソフトウェア割り込み</u>	<u>ファンクションリクエスト</u>
20H プログラムの終了	4CH プロセスの終了
27H プログラムをメモリにとどめ たまま終了	31H キーププロセス

1.8.1 ファイルコントロールブロック

V 2.0 以前のファイル管理のファンクションリクエストは、ファイルのファイルコントロールブロック (FCB) をアクセスします。このファイルコントロールブロックはファイル名、サイズ、レコード長、カレントレコードのポインタなどの情報を含んでいます。新しいハンドル形式のファンクションリクエストのファイル操作の大半を FCB 形式のファンクションリクエストでも実行できます。

プログラムセグメントプレフィクス内のオフセット 5CH と 6CH に 2 つの FCB のための空き領域が用意されています。FCB を取り扱う V 2.0 以前のシステムコールでは、“オープンされている FCB”、“オープンされていない FCB” のアドレスを指定のレジスタにセットします。オープンされていない FCB とは、ドライブ名とファイル名だけが入っているもので、ワイルドカード(*,?)を入れることができます。オープンされた FCB のすべてのフィールドは、オープンファイルシステムコール(ファンクション 0FH)によって埋められます。プログラムセグメントプレフィクスの説明と FCB の利用法については第 4 章を参照してください。表 1.13 に、FCB のフィールドの内容を示します。

表 1.13 ファイルコントロールブロック (FCB) のフィールド

フィールド名	大きさ (バイト)	オフセット	
		16 進	10 進
ドライブ番号	1	00H	0
ファイル名	8	01H~08H	1~8
拡張子	3	09H~0BH	9~11
カレント(現在の)ブロック	2	0CH, 0DH	12, 13
レコードサイズ	2	0EH, 0FH	14, 15
ファイルの大きさ	4	10H~13H	16~19
最後の書き込みが行われた日付	2	14H, 15H	20, 21
最後の書き込みが行われた時刻	2	16H, 17H	22, 23
予約域	8	18H~1FH	24~31
カレント(現在の)レコード	1	20H	32
相対レコード	4	21H~24H	33~36

1.8.2 FCBのフィールド

オフセット 00H:ドライブ番号

ディスクドライブを指定します。1はドライブA、2はドライブB、…を指定します。FCBをファイルの作成またはオープンのために使用するとき、このフィールドを0にセットするとカレントドライブ(省略時のドライブ)を指定することができます。オープンファイルシステムコール(ファンクション0FH)がこのフィールドを実際のドライブの番号にセットし直します。

オフセット 01H:ファイル名

8文字までの長さのファイル名がセットされます。ファイル名はフィールドの先頭から入り、8文字に満たない場合はスペースが入ります。予約されたデバイスファイル(PRNなど)を指定する場合、終結文字としてのコロン(:)をファイル名の最後に付けないでください。

オフセット 09H:ファイル名拡張子

3文字までの長さのファイル名拡張子がフィールドの先頭から入り、3文字に満たない場合はスペースが入ります。拡張子がない場合はすべてスペースになります。

オフセット 0CH:カレントブロック

現在のレコードが入っているブロック(128レコードが1単位)を示すポインタです。このカレントブロックフィールドとカレントレコードフィールド(オフセット 20H)によって、目的のレコードポインタを作成します。このフィールドは、オープンファイルシステムコールによって0にセットされます。

オフセット 0EH:レコードサイズ

バイト単位で表した論理レコードの長さをセットします。オープンファイルシステムコールによって、128がセットされます。レコード長が128バイトでない場合は、ファイルをオープンしたあと、このフィールドをセットしなければなりません。

オフセット 10H:ファイルのサイズ

バイト単位で表したファイルの大きさ、この4バイト分のフィールドの先頭のワードは、ファイルの大きさの下位の部分です。

オフセット 14H:最後に書き込みが行われた日付

ファイルが作成、更新された日付は以下のように2バイトにセットされます。

オフセット15H								オフセット14H							
15						9	8			5	4				0
Y	Y	Y	Y	Y	Y	Y	M	M	M	M	D	D	D	D	D
年							月				日				

オフセット 16H：最後の書き込みが行われた時刻

ファイルが作成、更新された時刻、分、秒は以下のように2バイトにセットされます。

オフセット 17H								オフセット 16H						
15				11	10			5	4					0
H	H	H	H	H	M	M	M	M	M	M	S	S	S	S
時					分					秒/2				

オフセット 18H：予約域

このフィールドは、MS-DOS が使用するために確保されています。

オフセット 20H：カレントレコード

現在のブロック内の128個の内の1つを示しています。前述のカレントブロックフィールド（オフセット 0CH）とこのカレントレコードフィールドによって、カレントレコードポインタが作成されます。なお、オープンファイルシステムコールは、このフィールドの初期値設定を行いません。このファイルに対してシーケンシャルなリード／ライトを行うためには、事前にこのフィールドをセットしておかなければなりません。

オフセット 21H：相対レコード

ファイルの先頭（0から始まる）からカウントした、現在選択されているレコード番号を示します。なお、オープンファイルシステムコールは、このフィールドの初期値設定を行いません。このファイルに対して、ランダムなリード／ライトを行うためには、事前にこのフィールドをセットしておかなければなりません。レコードサイズが64バイト未満の場合、フィールド全体の4バイトが、64バイト以上の場合、最初の3バイトのみが使用されます。

注意：プログラムセグメントプレフィクス内のオフセット 5CH のFCBを使用する場合、相対レコードフィールドの最終バイトは、オフセット80Hから開始するフォーマットされていないパラメータエリアの先頭バイトです。これは、デフォルトのディスク転送アドレスです。

1.8.3 拡張FCB

拡張FCBは、ディスクディレクトリ中で、特別なアトリビュート（属性）を持つファイルを作成・検索するのに使用されます。なお、通常のFCBの前の7バイトからなり、つぎのようにフォーマットされています。なお、アトリビュートバイトの詳細については1.5.6を参照してください。

フィールド名	大きさ (バイト)	オフセット (10 進)
フラグバイト (FFH) (拡張 FCB であることを示します、)	1	- 7
予約域	5	- 6
アトリビュートバイト	1	- 1

1.9 システムコールの使い方

第1章の残りの 1.9~1.11 は、アプリケーションプログラム上でのシステムコールの使い方、レジスタの指定も含めた各システムコールの詳細について述べます。

1.9.1 割り込みの使い方

MS-DOS は、システム自身の使用のために、20H から 3FH までの割り込みタイプを予約しており、80H~FCH に割り込みルーチンアドレステーブルを持っています。割り込みの多くはファンクションコールに置き換えられています。3つの MS-DOS 割り込みハンドラ(プログラムの終了、CTRL-C、致命的エラーによる中断)に対して、ユーザーがルーチンを作成するために、この3つの MS-DOS の割り込みハンドラについて、1.10 で解説しています。割り込みを行うには、レジスタに必要なデータを代入して、割り込みを実行します。

1.9.2 ファンクションリクエストの使い方

ファンクションリクエストは、システム資源の管理を行う MS-DOS のルーチン群をコールします。ファンクションリクエストをコールする標準シーケンス(手続き)はつぎのとおりです。

1. 必要とされるデータをそれぞれのレジスタにセットします。
2. ファンクション番号を AH にセットします。
3. 必要ならば、アクションコードを AL にセットします。
4. 割り込みタイプ 21H を実行します。

もし、プログラムが標準のプログラムセグメントプレフィクス (PSP) を持っている場合は、割り込みタイプ 21H を PSP 内のオフセット 50H をロングコールすることで代用できます。ただし、割り込みタイプ 21H の使用をお奨めします。

CP/M のシステムコール規約と一致したシーケンスも利用することができます。

1. 必要とされるデータをそれぞれのレジスタにセットします（標準シーケンスと同様）。
2. ファンクション番号を、CL レジスタにセットする。
3. 現在のコードセグメント内のロケーション 5 に対して、セグメント内コールを実行する。

この方法は、AL レジスタにパラメータを渡さない、ファンクション 00H から 24H までにおいてのみ、使用することができます。この方法によってファンクションコールが行われると AX レジスタの内容は常に失われてしまいます。

1.9.3 高級言語からのコール

アセンブリ言語モジュールとリンク可能な高級言語から、システムコールを行うことができます。他のコール方法をつぎに示します。

BASIC からのコール

システムコールを利用する場合、コンパイラとインタプリタでは異なった方法を使用します。コンパイルされたモジュールは、アセンブリ言語で開発されたモジュールとリンクして 1 つのプログラムとすることができます。インタプリタの場合は、CALL 文またはUSR 関数を使用します。

1.9.4 レジスタの処理

MS-DOS は、ファンクションリクエストをコールしたときに内部的にスタックを使います。そのために、リターン情報に使われていないレジスタの内容は、保存されます。しかし、プログラムのスタック領域の大きさを、割り込み処理を実行するのに十分な大きさ（少なくとも、他の処理に必要な大きさ+128 バイト）にしなければなりません。

1.9.5 エラー処理

エラーが起きたときに、新しいファンクションリクエスト（バージョン 2.0 以降）は、キャリーフラグをセットし、AX にエラーコードを返します。つぎの表は、エラーコードの一覧です。

コード	意 味	コード	意 味
1	無効なファンクションコード Invalid function code	16	カレントディレクトリを削除しようとした Attempt to remove the current directory
2	ファイル名が見つからない File not found	17	同じデバイスではない Not same device
3	パス名が見つからない Path not found	18	これ以上ファイルはない No more files
4	オープンファイル過多 Too many open files (no open handles left)	19	ディスクがライトプロテクト状態 Disk is write-protected
5	アクセスできない Access denied	20	ディスクユニット不良 Bad disk unit
6	無効なハンドル Invalid handle	21	ドライブが準備されていない Drive not ready
7	メモリコントロールブロック破損 Memory control blocks destroyed	22	無効なディスク コマンド Invalid disk command
8	メモリ不足 Insufficient memory	23	CRCエラー CRC error
9	無効なメモリブロックアドレス Invalid memory block address	24	無効な長さ Invalid length (disk operation)
10	無効な環境 Invalid environment	25	シークエラー Seek error
11	無効な書式 Invalid format	26	MS-DOSのディスクではない Not an MS-DOS disk
12	無効なアクセスコード Invalid access code	27	セクタが見つからない Sector not found
13	無効なデータ Invalid data	28	紙切れ Out of paper
14	予約 (使用されていない) RESERVED	29	書き込み失敗 Write fault
15	無効なドライブ名 Invalid drive	30	読み出し失敗 Read fault

コード	意 味	コード	意 味
31	通常の失敗 General failure	57	ネットワークアダプタのハード エラー Network adapter hardware error
32	シェアリング違反 Shareing violation	58	ネットワークからの不当な応答 Incorrect response from net work
33	ロック違反 Lock violation	59	予期できないネットワークエラー Unexpected network error
34	ディスク指定の失敗 Wrong disk	60	リモートアダプタが合致しない Incompatible remote adapter
35	FCB使用不可能 FCB unavailable	61	プリント待ち行列が一杯 Print queue full
36 49	予約 (使用されていない) RESERVED	62	待ち行列は一杯ではない Queue not full
50	ネットワークリクエストが準備さ れていない Network request not supported	63	プリントファイルのためのスペー スが不十分 Not enough space for print file
51	リモートコンピュータが LISTEN していない Remote computer not listening	64	ネットワーク名は既に削除されて いる Network name was deleted
52	ネットワーク名の2重定義 Duplicate name on network	65	アクセスできない Access denied
53	ネットワーク名が見つからない Network name not found	66	ネットワークデバイスのタイプが 不当 Network device type incorrect
54	ネットワークビジー Network busy	67	ネットワーク名が見つからない Network name not found
55	ネットワークデバイスはこれ以上 ない Network device no longer exists	68	ネットワーク名の限界を超えた Network name limit exceeded
56	ネットワーク BIOSの限界を超え た Net BIOS command limit ex ceeded		

コード	意 味	コード	意 味
69	ネットワーク BIOS セッション の限界を超えた Net BIOS session limit exceed ed	80	ファイルが存在する File exists
70	一時休止 Temporarily paused	81	予約 RESERVED
71	ネットワークの要求が受けつけら れない Network request not accepted	82	作成不能 Cannot make
72	プリンタ、ディスクのリダイレク ション休止 Print or disk redirection is paused	83	割り込みタイプ24Hの失敗 Interrupt 24H failure
73 79	予約（使用されていない） RESERVED	84	ストラクチャの不良 Out of structures
		85	割り当て済み Already assigned
		86	無効なパスワード Invalid password
		87	無効なパラメータ Invalid parameter
		88	ネットワークへの書き込み失敗 Net write fault

エラー処理は、各コールのすぐ後につぎのようなステートメントを置きます。

JC <エラー処理ルーチンラベル>

アプリケーションで、AX の内容を調べて、エラーの内容を判別し、処理ルーチンへ制御を移します。

V 2.0 以前のシステムコールのいくつかは、システムコールが成功したかどうかをレジスタに返します。この場合は、アプリケーションが、そのエラーコードをチェックし、適切な処理を行ってください。

拡張されたエラーコード

MS-DOS の新しいバージョンでは、古いシステムコールでは使われなかったさらに詳細なエラーメッセージが加わっていますが、互換性を保つために古いエラーメッセージとは区別されます。

これらの新しいエラーメッセージは、ファンクションリクエスト 59H（拡張されたエラーを

得る)で得られ、MS-DOS が返す重大なエラーコードの大半を網羅しています。また、ファンクション 59H の項では、新しく詳細なエラーコードの一覧と、このファンクションリクエストの使い方が解説されております。

1.9.6 システムコールの解説方法

各システムコールの項では必要に応じて、その機能の概要(機能)、実行前にセットするレジスタとその概要(コール)、実行後に返されるレジスタとその概要(リターン)、コールとリターンで使用するレジスタの詳細とシステムコールの解説(解説)、マクロ定義の例(マクロ定義)、そのマクロを使ったプログラミング例(例)が解説されます。

図1に、システムコールの各種レジスタの状態の例としてファンクション 27H (ランダムブロックリード)の一部、図2にシステムコールの使い方としてのサンプルプログラムとその解説を示します。

INT 21H			
Random Block Read		ファンクション 27H	
機 能	ランダムブロックリード	AX	AH AL
コール	AH = 27H DS : DX オープンされた FCB CX 読み出すべきレコード数	BX	BH BL
リターン	AL = 00H 読み出しの正常終了 01H EOF, 空レコード 02H セグメントの終わり 03H EOF, レコードの一部分 CX 読み取られたレコード数	CX	CH CL
		DX	DH DL
			SP
			BP
			SI
			DI
			IP
			FLAGSH FLAGSL
			CS
			DS
			SS
			ES

図1 システムコールの説明の例

サンプルプログラム

つぎのサンプルプログラムは、システムコールの使い方とデータの宣言だけから構成されています。このサンプルプログラムは、セグメントの宣言や MS-DOS への戻り方といったプログラムを作る上での基本的な事が含まれています。なおこのサンプルプログラムは COM 形式のファイルとして実行されるものです。

code	segment		
	assume	cs: code, ds: code, es: nothing, ss: nothing	
	org	100H	
start:	jmp	begin	
;			
filename	db	"b: ¥textfile. asc", 0	
buffer	db	129 dup (?)	
handle	dw	?	
;			
begin:	open_handle	filename, 0	; ファイルのオープン
	jc	error_open	; エラー処理へ
	mov	handle, ax	; ハンドルをセーブ
read_line:	read_handle	handle, buffer, 128	; 128 バイトを読み込む
	jc	error_read	; エラー処理へ
	cmp	ax, 0	; ファイルエンドか?
	je	return	; はいの時, 処理終了 return へ
	mov	bx, cx	; いいえの時, 読み出すべき
			; レコード数をセット
	mov	buffer[bx], "\$"	; 終了ストリングのセット
	display	buffer	; ストリングの表示(09H)
	jmp	read_line	; 128 バイト読み込む処理を継続
return:	end_process	0	; 処理終了し MS-DOS へ戻る
last_inst:			; メッセージ表示
;			; プログラムの終了
code	ends		
	end	start	

図2 システムコールにおけるプログラムの例

このサンプルプログラムでは、システムコールの使い方をマクロ定義にしてあります。マク

ロ定義 (open-handle, read-handle, display, end-process) については、第1章の最後にあるマクロ定義例を参照してください。

これらマクロは、第4章で解説されている COM 形式のプログラムのための環境を想定しています。特別な例として、同じ値のすべてのレジスタを定義する場合があります。通常、マクロはレジスタの保護も、メインコードからエラー処理ルーチンに行くときのラベルのチェックも行いません。それらはマクロ定義のサブルーチンを小さくするために、マクロをコールするメインのアセンブルプログラムで定義します。

サンプルプログラムでのエラー処理

システムコールがエラーコードを返したときに、サンプルプログラムはエラー状態をチェックし、エラー処理ルーチンへ移ります(エラールーチンの内容は省略します)。ふつう、エラー処理ルーチンは簡単なメッセージを表示するだけで、作業を続行します。しかし重大なエラーが起きたときは、メッセージを表示し、プログラムを終了します(ただし、ファイルをクローズするなどの処理を行います)。

以下に各割り込みタイプとファンクションコールを数字順に解説します。

1.10 割り込み

プログラムの使用できる割り込みタイプ 20H~27H について解説します。

表 1.14 MS-DOS 割り込み

16 進	10 進	機能
20H	32	プログラムの終了
21H	33	ファンクションリクエスト
22H	34	終了アドレス
23H	35	〈CTRL-C〉の抜け出しアドレス
24H	36	致命的エラーによる打ち切りアドレス
25H	37	アブソリュートディスクリード
26H	38	アブソリュートディスクライト
27H	39	プログラムをメモリにとどめたまま終了
28~3FH	40~63	予約

注意：

各ファンクションコールの解説にあるサンプルプログラムは、参考のために記載しているものです。これらのサンプルプログラムは、そのままでは動作しません。

Program Terminate

割り込みタイプ 20H

機能 プログラムの終了

コール CS プログラムセグメントプレフィックスの、セグメントアドレス

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 割り込みタイプ 20H によって現在のプロセスが終了し、制御が親プロセスに戻ります。すべてのオープンされているファイルをクローズします。V 2.0 以前の MS-DOS では、COM ファイルの終了は、ほとんどこの割り込みで行われます。

この割り込みをかけるためには、その前に CS レジスタ内にプログラムセグメントプレフィックスのセグメントアドレスを入れておきます。

以下の抜け出しアドレスは、プログラムセグメントプレフィックスを回復するものです。

抜け出しアドレス	オフセット
プログラムの終了	0AH
CTRL-C	0EH
重大なエラー	12H

すべてのファイルバッファの内容は、すべてディスクに書き出されます。

注意：

この割り込みをかける前にサイズを変更したすべてのファイルをクローズしておいてください。変更されたファイルがクローズされていない場合、そのファイルの大きさはディレクトリに正しく書き込まれません。ファイルクローズについては、ファンクション 10H, 3EH を参照してください。

割り込みタイプ 20H は、MS-DOS V 2.0 以前と互換性があります。V 3.1 以降で開発する新規のプログラムでは、ファンクションリクエスト 4CH を使用して、プロセスを終了するとよいでしょう。

マクロ定義

```
terminate macro
```

```
    int 20H
```

```
endm
```

例

この例は、スクリーンにメッセージを表示し、MS-DOS に戻るプログラムです。1.9.6 のサンプルプログラムも参照してください。

```
message    db "displayed by INT20H example", 0DH, 0AH, "$"
```

```
;
```

```
int_20H:   display message    ;メッセージ表示(09H)
```

```
           terminate          ;プログラムの終了
```

```
code       ends
```

```
end       start
```

Function Request

割り込みタイプ 21H

機 能	ファンクションリクエスト
コール	AH ファンクションリクエストの番号 他のレジスタ 個々のファンクションで指定された内容
リターン	各ファンクションの解説を参照

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解 説 AH レジスタには、目的のシステムファンクションの番号をセットします。詳細については、1.11 章の“ファンクションリクエスト”を参照してください。

マクロ定義 本章で解説しているすべてのファンクションのマクロ定義に割り込みタイプ 21H が入っているので、この割り込みのためのマクロは特に定義しません。

例 時刻を得るファンクションのコールを行います。

```
mov ah, 2CH      ;時刻を得るファンクション2CHをコール
int 21H          ;ファンクションリクエスト
```


Terminate Address	割り込みタイプ 22H
CTRL-C Exit Address	23H
Critical Error Handler Address	24H

機 能	終了アドレス (割り込みタイプ 22H)
	〈CTRL-C〉の抜け出しアドレス (割り込みタイプ 23H)
	致命的エラーによる中断アドレス (割り込みタイプ 24H)

これらは真の割り込みではなく、セグメントおよびオフセットアドレスのための記憶域の位置であり、指定された環境下で MS-DOS によって、割り込みがかけられます。ユーザー自身の割り込みハンドラを作成したい場合、ファンクションリクエスト 35H (割り込みベクタを得る) を使ってアドレスを得て、次にファンクションリクエスト 25H (割り込みベクタのセット) を使用して、セットします。

割り込みタイプ 22H … 終了アドレス

プログラムが終了するとき、MS-DOS はベクタテーブルの割り込みタイプ 22H のエントリアドレスに制御を移行します。このアドレスは、MS-DOS がプログラムセグメントを作成したときは、プログラムセグメントプレフィクス内のオフセット 0AH にコピーされます。

割り込みタイプ 23H … 〈CTRL-C〉の抜け出しアドレス

〈CTRL-C〉を入力した場合、MS-DOS はベクタテーブルの割り込みタイプ 23 H のエントリアドレスに制御を移行します。このエントリアドレスは、MS-DOS がプログラムセグメントプレフィクスを作成したときは、プログラムセグメントプレフィクス内のオフセット 0EH にコピーされます。

ユーザー独自の〈CTRL-C〉ルーチンを作成する場合、以下の点に注意してください。

〈CTRL-C〉ルーチンですべてのレジスタの内容を保存すれば、IRET 命令でこのルーチンを終了してプログラムを継続することができます。割り込み発生時、すべてのレジスタの内容は MS-DOS がコールされたときの値にセットされます。IRET で戻るときにレジスタの値を保存するかぎり、MS-DOS のシステムコールの使用を含めて、〈CTRL-C〉ルーチンに制限はありません。

〈CTRL-C〉ルーチンは、ロングリターン (Far RET) を使うことによりキャリーフラグを用いて割り込み発生前のプログラムを強制終了するか、続行するかを決定することができます。MS-DOS はキャリーフラグがセットされていれば、プログラムを強制終了させ、そうでなければ、IRET によって戻ったときと同様にプログラムを続行します。

プログラムがファンクションリクエスト 09 H, 0AH, パッファード I/O のいずれかを実行中に <CTRL-C> によりユーザーが作成した <CTRL-C> ルーチンに割り込んだとき, IRET でプログラムを続行させることにより, 入出力は次の行の先頭から再開されます。

プログラムがファンクション 4B00 H (プログラムのロードと実行) を使うなどして第 2 のプログラムセグメントプレフィクスをつくり, ベクタテーブルの <CTRL-C> のアドレスを変更する第 2 のプログラムを実行した場合, MS-DOS は, 第 1 のプログラムに制御が戻る前に <CTRL-C> のアドレスを第 2 のプログラムの実行前の値に戻します。

注意:

MS-DOS は, INT 23 H を実行する場合必ず画面に " ^C" と 0DH, 0AH (キャリッジリターン, ラインフィード) を出力しますが, これを取り消すことはできません。

割り込みタイプ 24H …致命的エラーによる中断アドレス

I/O ファンクションコールの内の 1 つを実行中に致命的ディスクエラーが発生した場合, MS-DOS はベクタテーブルの割り込みタイプ 24H のエントリアドレスに制御を移行します。このアドレスは, MS-DOS がプログラムセグメントを作成したときは, プログラムセグメントプレフィクス内のオフセット 12H にコピーされます。

割り込みタイプ 25H (アブソリュートディスクリード) または割り込みタイプ 26H (アブソリュートディスクライト) を実行中にエラーが発生した場合, 割り込みタイプ 24H は実行できません。これらのエラーは, 通常 COMMAND.COM 内の MS-DOS エラールーチンによって処理されます。このルーチンによってディスクアクセスの再試行が行われ, つぎにユーザーはこの動作を打ち切るか, または再試行するか, またはエラーを無視して続行することができます。つぎの項目は, 割り込みタイプ 24H ルーチンに必要な条件, エラーコード, レジスタとスタックの管理について解説します。

1.10.1 エントリの状態

I/O エラーに対して, 3 回再試行した後に MS-DOS は割り込みタイプ 24H を実行し, 割り込み処理ルーチンは割り込みタイプ 24H から制御を渡されます。AX と DI レジスタにはエラーについての情報が入ります。BP には, エラーを起こしたデバイスについて記述されているデバイスヘッダコントロールブロックのオフセットが入ります (セグメントアドレスは, SI に入ります)。

1.10.2 割り込みタイプ 24H ハンドラの必要条件

プログラムを「中止するか, 再試行するか, 無視するか」の選択をさせるプロンプトを表示する MS-DOS の割り込みタイプ 24H の処理ルーチンを使いたい場合, ユーザーのエラー処理ルーチンはフラグをブッシュし, 標準的な割り込みタイプ 24H ハンドラのアドレスを FAR コ

ールします(割り込みタイプ 24H のベクタを変更したユーザーのプログラムは、そのベクタアドレスをセーブしておかなくてはなりません)。ユーザーが前述のプロンプトに答えると、MS-DOS はユーザーのプログラムに制御を戻します。

ユーザーの割り込みハンドラでは、他の処理を行う前に、BX, CX, DX, ES, DS, SS, SP の内容を保存しなくてはなりません。使えるファンクションコールは 01H~0CH, 59H だけです(もし、他のファンクションコールを使用した場合、MS-DOS のスタック領域はこわされ、その後の動作は保証されません)。また、デバイスヘッダの内容を変えてはいけません。

注意:

ユーザーが作成したアプリケーションプログラムでスタック領域がこわされる問題が起きたときは、スタックフレームを変更してみるのもよいでしょう。

もし割り込みタイプ 24H ルーチンから MS-DOS に返らずにユーザーのプログラムに戻る場合は、アプリケーションプログラムのレジスタをリストアし、スタックの最後の 3 ワードだけを残して、IRET を行います。エラーが生じた I/O ファンクションリクエストから直ちにプログラムに戻ります。この処理を行った場合、MS-DOS は 0CH 以上のファンクションコールが行われるまで、不安定な状態となります。

エラーコード

AX が返すディスクエラーコード

AH のビット 7 が 0 の場合、ディスクドライブ関連のエラーであることを示します。AL はエラーを起こしたドライブの番号 (0=A, 1=B……) です。AH のビット 0 は、エラーが起きたのが書き込み時か、読み込み時かを示します (0 = 読み込み時, 1 = 書き込み時)。AH のビット 1 と 2 は、エラーを起こしたディスクの領域の種類を示します。その内容をつぎに示します。

ビット

2-1 種類

- | | |
|----|-----------------------|
| 00 | MS-DOS 領域 |
| 01 | ファイルアロケーションテーブル (FAT) |
| 10 | ディレクトリ |
| 11 | データ領域 |

AH のビット 3~5 は、エラープロンプトに対する有効な返答を指定します。その内容をつぎに示します。

ビット	内容	返答
3	0	プログラムの失敗可
	1	プログラムの失敗不可
4	0	再試行可
	1	再試行不可
5	0	エラーの無視可
	1	エラーの無視不可

もし、再試行が不可の場合、MS-DOS は再試行をせずに失敗したとみなします。もし、エラーを無視するのを不可にした場合、MS-DOS はエラーを無視せずに失敗したとみなします。もし、プログラムの失敗を不可にした場合、MS-DOS はプログラムを中止します。プログラムの中止は常に可になっています。

AX が返す他のデバイスのエラーコード

もし、AH ビット 7 が 1 の場合、ファイルアロケーションテーブル (FAT) のメモリーイメージが悪いか、キャラクタデバイスにエラーがあることを示します。BP:SI によって指定されるデバイスヘッダはデバイスとエラーの種類を示す属性を表す 1 ワードを含んでいます。

属性を表す 1 ワードは、デバイスヘッダのオフセット 04H にあります。ビット 15 はデバイスの種類を示します (0=ブロック, 1=キャラクタ)。

ビット 15 が 0 (ブロックデバイス) の場合、FAT のメモリーイメージにエラーの原因があります。

ビット 15 が 1 (キャラクタデバイス) の場合、キャラクタデバイスにエラーの原因があります。DI がエラーコードを示し、AL のビット 0~3 は、エラーを起こしたキャラクタデバイスの種類を示します。その内容をつぎに示します。

ビット	意味
0	標準入力
1	標準出力
2	NUL デバイス
3	クロックデバイス

デバイスヘッダコントロールブロックの詳細については 2 章を参照してください。

DI が返すエラーコード

DI の下位バイトはエラーコードを示します。その内容をつぎに示します。
また、上位バイトは不定です。

エラーコード 意味

00H	ライトプロテクトされているディスクに書き込みを行おうとした
01H	存在しないユニット
02H	ドライブの準備ができない
03H	存在しないコマンド
04H	データの CRC エラー
05H	バッドドライブリクエストストラクチャの長さ
06H	シークエラー
07H	存在しないメディアタイプ
08H	セクタが存在しない
09H	プリンタの用紙切れ
0AH	書き込み不良
0BH	読み込み不良
0CH	一般的なディスク不良

ユーザーの用意した割り込みタイプ 24H ハンドラは、ファンクション 59H（拡張されたエラーコードを得る）を実行することで、詳細なエラーの情報を得ることができます。

スタックの内容は、以下のとおりです。

スタックの一番上→	IP	INT 24H が出た時点での MS-DOS のレジスタ (致命的エラーによる割り込み)
	CS	
	FLAGS	
	AX	} INT 21H が出た時点でのユーザーレジスタ
	BX	
	CX	
	DX	
	SI	
	DI	
	BP	
	DS	
	ES	
	IP	
	CS	ユーザーから DOS への割り込み
	FLAGS	

再試行（リトライ）

レジスタには、動作の再試行を行うために必要とされるデータが入っています。AL に以下の値の 1 つを入れ、IRET を実行することによって、行うべき動作を指定します。

- 00H エラーを無視する
- 01H 再試行
- 02H プログラムを打ち切る
- 03H プログラム上のシステムコールの失敗

エラーを無視するオプションを指定する場合は、MS-DOS がエラーが生じていないと判断して処理するために、予期せぬ状態になることが考えられますので、注意してください。

再試行を指定する場合は、レジスタの内容を変更しないでください。

Absolute Disk Read

割り込みタイプ 25H

機 能	アブソリュートディスクリード
コール	AL ドライブ番号 DS: BX ディスク転送アドレス (DTA) CX 読み込みセクタ数 DX 読み込み開始相対セクタ番号

リターン	CF = 1 エラー発生 AL にエラーコードを返す 0 処理の正常終了
-------------	--

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説	レジスタには、以下のものが入っていなければなりません。
------------	-----------------------------

- AL ドライブ番号 (00H=A, 01H=B, ...)
- BX ディスク転送アドレスのオフセット (DS 内のセグメントアドレスから)
- CX 読み込むべきセクタ数
- DX 読み込み開始相対セクタ番号

警告: このシステムコールの使用はできるだけ避けるようにしてください。ファイルのアクセスは、通常ファンクションコールを通して行ってください。アブソリュートディスク I/O の、MS-DOS 上位のバージョンに対する互換性は保証されません。

この割り込みによって、制御は直接 MS-DOS のデバイスドライバに移行します。CX で指定した数のセクタが、ディスクからディスク転送アドレスに読み込まれます。この割り込みの使い方や処理は、データが書き出されるのではなく読み込まれるということを除けば、割り込みタイプ 26H と同一です。なお、この割り込みは、使い方を誤ると動作が不安定になりますので注意してください。

注意:

セグメントレジスタ以外のすべてのレジスタの内容は、このコールによって破壊されるので、割り込みをかける前にユーザーのプログラムで使用するすべての

レジスタの内容を必ず保存しておいてください。

このコールを行うとき、フラグはスタックに積まれ（プッシュフラグ：PUSHF）、システムによって終了後もそこに残っています（処理の結果を表すデータがフラグ内に返されるためです）。スタックが制限なく増加することを防ぐために、終了後必ずそのスタックを取り出し（ポップフラグ：POPF）てください。

ディスク動作が正しく行われた場合、キャリーフラグ(CF)=0に、正しく行われなかった場合、CF=1になり、ALにエラーコードが返されます。（エラーコードとその意味については、割り込みタイプ 24H を参照してください。）

マクロ定義

```
abs-disk-read macro disk, buffer, num-sectors, start
    mov     al, disk
    mov     bx, offset buffer
    mov     cx, num-sectors
    mov     dx, start
    int     25H
    popf
endm
```

例

つぎのプログラムは、ドライブAの片面ディスクの内容をドライブBのディスクにコピーするものです。このプログラムでは、32K バイトの大きさのバッファが使用されています。

```
prompt db "Source in A, target in B", 13, 10
       db "Any key to start. $"
start  dw 0
buffer db 64 dup (512 dup (?)) ; 64 sectors
;
int_25H: display prompt          ; prompt の内容を表示(09H)
         read_kbd                ; キーボード入力待ち(08H)
         mov     cx, 5            ; 1回(64セクタ)の読み込み回数(5)をセット
copy:   push     cx              ; 読み込みカウンタ(回数)をセーブ
         abs-disk-read 0, buffer, 64, start ; アブソリュートディスクリード
         abs-disk-write 1, buffer, 64, start ; アブソリュートディスクライト(26H)
         add     start, 64        ; 次の64セクタについて行う
         pop     cx              ; 読み込みカウンタをリストア
         loop    copy
```

Absolute Disk Write

割り込みタイプ 26H

機 能	アブソリュートディスクライト
コール	AL ドライブ番号 DS: BX ディスク転送アドレス (DTA) CX 書き出しセクタ数 DX 書き出し開始相対セクタ番号

リターン	CF = 1 エラー発生 AL にエラーコードを返す 0 処理の正常終了
-------------	--

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解 説 レジスタには、以下のものが入っていなければなりません。

- AL ドライブ番号 (00H=A, 01H=B, ...)
- BX ディスク転送アドレスのオフセット (DS 内のセグメントアドレスから)
- CX 書き出すべきセクタ数
- DX 書き出し開始相対セクタ番号

警告: このシステムコールの使用はできるだけ避けるようにしてください。ファイルのアクセスは、通常のファンクションコールを通して行ってください。アブソリュートディスク I/O の、上位バージョンに対する互換性は保証されません。

この割り込みによって、制御が直接に MS-DOS の BIOS に移行します。CX で指定されたセクタ数が、ディスク転送アドレスからディスクに書き出されます。この処理は、データがディスクから読み取られるのではなくディスクに書き込まれるということを除けば、割り込みタイプ 25H と同一です。また、この割り込みは、使い方を誤ると動作が不安定になりますので注意してください。

注意:

セグメントレジスタ以外のすべてのレジスタの内容は、このコールによって破壊されるので、割り込みをかける前にユーザーのプログラムで使用するすべてのレジスタを、必ず保存しておいてください。

このコールを行うとき、フラグはシステムによってスタックに積まれ（プッシュフラグ：PUSHF）終了後もそこに残っています（処理の結果を表すデータがフラグ内に返されるためです）。スタックが制限なく増加することを防ぐために、終了後必ずそのスタックを取り出し（ポップフラグ：POPF）てください。

ディスク動作が正しく行われた場合、キャリーフラグ(CF)=0に、正しく行われなかった場合、CF=1になり、ALにエラーコードが返されます。（エラーコードとその意味については、割り込みタイプ 24H を参照してください。）

マクロ定義

```
abs-disk-write macro disk, buffer, num-sectors, start
    mov     al, disk
    mov     bx, offset buffer
    mov     cx, num-sectors
    mov     dx, start
    int     26H
    popf
endm
```

例

つぎのプログラムは、ドライブAの片面ディスクの内容をドライブBのディスクにコピーし、書き出し(ライト)が行われるごとにベリファイ、検証を行うものです。このプログラムでは、32K バイトの大きさのバッファが使用されています。

```
off      equ 0
on       equ 1
;
prompt   db "Source in A,  target in B", 13, 10
         db "Any key to start,  $"
start    dw 0
buffer   db 64 dup (512 dup (?)) ; 64 sectors
;
int_26H: display prompt      ; prompt の内容を表示(09H)
         read_kbd           ; キーボード入力待ち(08H)
         verify on          ; ベリファイフラグをONにする(2EH)
         mov     cx, 5       ; 1回(64セクタ)の読み込み回数(5)をセット
copy:    push     cx         ; 書き出しカウンタ(回数)をセーブ
         abs-disk-read 0, buffer, 64, start ; アブソリュート
                                                ; ディスクリード(25H)
         abs-disk-write 1, buffer, 64, start ; アブソリュート
                                                ; ディスクライト
```


add	start, 64	; 次の64セクタについて行う
pop	cx	; 書き出しカウンタをリストア
loop	copy	
verify	off	; ベリファイフラグをoffにする(2EH)

Terminate But Stay Resident

割り込みタイプ 27H

機能	プログラムをメモリにとどめたまま終了
コール	CS : DX コードの最終バイトのつぎに来る最初のバイトのアドレス
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解説 このコールは、64Kバイト以下のプログラムをメモリにとどめたまま終了します。このコールは、デバイススペシフィック割り込みハンドラでよく使用されます。

この割り込みは、V 2.0 以前の MS-DOS と互換性を保つために用意されています。新規のプログラムでは、ファンクション 31H（キーププロセス）を使用するとよいでしょう。このファンクションは 64K バイトを超えるプログラムでもメモリにとどめることができます。ユーザーが作ったプログラムが V 2.0 以前の MS-DOS に対する互換性を要求されない限り、常駐するプログラムにリターン情報を渡すことができます。

DX には、常駐させるプログラムコードの最終バイトの次に来る先頭のオフセット（CS のセグメントアドレスからの）が入っていなければなりません。割り込みタイプ 27H が実行されたときに、プログラムは終了し、制御は MS-DOS に戻ります。しかし、他のプログラムによるオーバーレイは行われません。ファイルはオープンされたまま、クローズされていません。割り込みが実行されたとき、CS には必ずプログラムセグメントプレフィクス（割り込みが実行されたときの ES と DS の値）のセグメントアドレスが入っていなければなりません。

この割り込みは、EXE 形式のプログラムで使用することはできません。またこの割り込みは割り込みタイプ 22H、23H、24H のベクタをリストアしますので、

新しい〈CTRL-C〉や致命的エラーのエラーハンドラを作ることができません。

マクロ定義

```
stay_resident macro last_instruc
                mov     dx, offset last_instruc
                inc     dx
                int      27H
                endm
```

例

```
; CS のセグメントアドレスは割り込み実行時の PSP 値(ES と DS の値)と
; 同じでなければならない
mov     DX, LastAddress
int     27H
; この割り込みにリターン情報はない
```

1.11 ファンクションリクエスト

以下にファンクションリクエスト 00H~62H について解説します。

表 1.15 に以下に解説するファンクションリクエストを数字の順に、表 1.16 に 50 音順に示します。

表 1.15 MS-DOS ファンクションリクエスト，番号順

番号	ファンクション名	
00H	プログラムの終了	Terminate Program
01H	キーボード入力とエコー	Read Keyboard and Echo
02H	文字のスクリーン出力	Display Character
03H	補助入力	Auxiliary Input
04H	補助出力	Auxiliary Output
05H	文字のプリンタ出力	Print Character
06H	直接コンソール I/O	Direct Console I/O
07H	直接コンソール入力	Direct Console Input
08H	キーボード入力	Read Keyboard
09H	ストリングのスクリーン出力	Display String
0AH	バッファードキーボード入力	Buffered Keyboard Input
0BH	キーボードステータスの検査	Check Keyboard Status
0CH	バッファを空にしてキーボード入力	Flush Buffer, Read Keyboard
0DH	リセットディスク	Reset Disk
0EH	ディスクの選択	Select Disk
0FH	ファイルのオープン	Open File
10H	ファイルのクローズ	Close File
11H	最初のエントリを検索	Search for First Entry
12H	つぎのエントリを検索	Search for Next Entry
13H	ファイルの削除	Delete File
14H	シーケンシャルリード	Sequential Read
15H	シーケンシャルライト	Sequential Write
16H	ファイルの作成	Create File
17H	ファイル名の変更	Rename File
19H	カレントディスクを得る	Get Current Disk
1AH	ディスク転送アドレスのセット	Set Disk Transfer Address
1BH	デフォルトドライブのデータを得る	Get Default Drive Data
1CH	ドライブのデータを得る	Get Drive Data
21H	ランダムリード	Random Read

22H	ランダムライト	Random Write
23H	ファイルの大きさを得る	Get File Size
24H	相対レコードのセット	Set Relative Record
25H	割り込みベクタのセット	Set Interrupt Vector
26H	新しい PSP を作成する	Create New PSP
27H	ランダムブロックリード	Random Block Read
28H	ランダムブロックライト	Random Block Write
29H	ファイル名の解析	Parse File Name
2AH	日付を得る	Get Date
2BH	日付をセットする	Set Date
2CH	時刻を得る	Get Time
2DH	時刻をセットする	Set Time
2EH	ベリファイフラグのセット/リセット	Set/Reset Verify Flag
2FH	ディスク転送アドレスを得る	Get Disk Transfer Address
30H	MS-DOS バージョン番号を得る	Get MS-DOS Version Number
31H	キーププロセス	Keep Process
33H	〈CTRL-C〉検査のセット/リセット	Control-C Check
35H	割り込みベクタを得る	Get Interrupt Vector
36H	ディスクのフリースペースを得る	Get Disk Free Space
38H	国別情報を得る	Get Country Data
38H	国別情報をセットする	Set Country Data
39H	ディレクトリの作成	Create Directory
3AH	ディレクトリの削除	Remove Directory
3BH	カレントディレクトリの変更	Change Current Directory
3CH	ハンドルの作成	Create Handle
3DH	ハンドルのオープン	Open Handle
3EH	ハンドルのクローズ	Close Handle
3FH	リードハンドル	Read Handle
40H	ライトハンドル	Write Handle
41H	ディレクトリエントリの削除	Delete Directory Entry
42H	ファイルポインタの移動	Move File Pointer
43H	アトリビュート(属性)を得る/セットする	Get/Set File Attributes
4400H	IOCTL データを得る	Get IOCTL Data
4401H	IOCTL データをセットする	Set IOCTL Data
4402H	IOCTL キャラクタを受け取る	Receive IOCTL Character
4403H	IOCTL キャラクタを送る	Send IOCTL Character
4404H	IOCTL ブロックを受け取る	Receive IOCTL Block
4405H	IOCTL ブロックを送る	Send IOCTL Block

4406H	デバイスの入力ステータスのチェック	Get Input IOCTL Status
4407H	デバイスの出力ステータスのチェック	Get Output IOCTL Status
4408H	IOCTL の交換性	IOCTL Is Changeable
4409H	IOCTL リダイレクトブロック	IOCTL Is Redirected Block
440AH	IOCTL リダイレクトハンドル	IOCTL Is Redirected Handle
440BH	IOCTL リトライ	IOCTL Retry
440CH	一般 IOCTL (ハンドル用)	Generic IOCTL (for handles)
440DH	一般 IOCTL (ブロックデバイス用)	Generic IOCTL (for block devices)
440EH	論理ドライブマップの取得	Get Logical Drive Map
440FH	論理ドライブマップの設定	Set Logical Drive Map
45H	ファイルハンドルの二重化	Duplicate File Handle
46H	ファイルハンドルの強制二重化	Force Duplicate File Handle
47H	カレントディレクトリを得る	Get Current Directory
48H	メモリの割り当て	Allocate Memory
49H	割り当てられたメモリの解放	Free Allocated Memory
4AH	割り当てられたメモリブロックの変更	Set Block
4B00H	プログラムのロードと実行	Load and Execute Program
4B03H	オーバーレイのロード	Load Overlay
4CH	プロセスの終了	End Process
4DH	子プロセスからリターンコードを得る	Get Return Code Child Process
4EH	最初に一致するファイル名の検索	Find First file
4FH	つぎに一致するファイル名の検索	Find Next File
54H	ベリファイの状態を得る	Get Verify State
56H	ディレクトリエントリの変更	Change Directory Entry
57H	ファイルの日付/時刻を得る/セットする	Get/Set Date/Time of File
58H	アロケーションストラテジを得る/セットする	Get/Set Allocation Strategy
59H	拡張されたエラーコードを得る	Get Extended Error
5AH	一時ファイルの作成	Create Temporary File
5BH	新しいファイルの作成	Create New File
5C00H	ファイルアクセスのロック	Lock
5C01H	ファイルアクセスのロック解除	Unlock
5E00H	マシン名を得る	Get Machine Name
5E02H	プリンタセットアップ	Printer Setup
5F02H	割り当てリストのエントリを得る	Get Assign List Entry
5F03H	割り当てリストのエントリ作成	Make Assign List Entry
5F04H	割り当てリストのエントリのキャンセル	Cancel Assign List Entry
62H	PSP を得る	Get PSP

表 1.16 MS-DOS ファンクションリクエスト, 50 音順

ファンクション名		番号
IOCTL キャラクタを受け取る	Receive IOCTL Character	4402H
IOCTL キャラクタを送る	Send IOCTL Character	4403H
IOCTL データを得る	Get IOCTL Data	4400H
IOCTL データをセットする	Set IOCTL Data	4401H
IOCTL ブロックを受け取る	Receive IOCTL Block	4404H
IOCTL ブロックを送る	Send IOCTL Block	4405H
IOCTL の交換性	IOCTL Is Changeable	4408H
IOCTL リダイレクトハンドル	IOCTL Is Redirected Handle	440AH
IOCTL リダイレクトブロック	IOCTL Is Redirected Block	4409H
IOCTL リトライ	IOCTL Retry	440BH
新しい PSP を作成する	Create New PSP	26H
新しいファイルの作成	Create New File	5BH
アトリビュート(属性)を得る/セットする	Get/Set File Attributes	43H
アロケーションストラテジを得る/セットする	Get/Set Allocation Strategy	58H
一時ファイルの作成	Create Temporary File	5AH
一般 IOCTL (ハンドル用)	Generic IOCTL (for handles)	440CH
一般 IOCTL (ブロックデバイス用)	Generic IOCTL (for block device)	440DH
MS-DOS バージョン番号を得る	Get MS-DOS Version Number	30H
オーバーレイのロード	Load Overlay	4B03H
拡張されたエラーコードを得る	Get Extended Error	59H
カレントディスクを得る	Get Current Disk	19H
カレントディレクトリを得る	Get Current Directory	47H
カレントディレクトリの変更	Change Current Directory	3BH
キーププロセス	Keep Process	31H
キーボードステータスの検査	Check Keyboard Status	0BH
キーボード入力	Read Keyboard	08H
キーボード入力とエコー	Read Keyboard And Echo	01H
国別情報を得る	Get Country Data	38H
国別情報をセットする	Set Country Data	38H
子プロセスからリターンコードを得る	Get Return Code Child Process	4DH
<CTRL-C> 検査のセット/リセット	Control-C Check	33H
最初に一致するファイル名の検索	Find First File	4EH
最初のエントリを検索	Search For First Entry	11H
デバイスの出力ステータスのチェック	Get Output IOCTL Status	4407H

デバイスの入力ステータスのチェック	Get Input IOCTL Status	4406H
シーケンシャルライト	Sequential Write	15H
シーケンシャルリード	Sequential Read	14H
時刻を得る	Get Time	2CH
時刻をセットする	Set Time	2DH
ストリングのスクリーン出力	Display String	09H
相対レコードのセット	Set Relative Record	24H
直接コンソール I/O	Direct Console I/O	06H
直接コンソール入力	Direct Console Input	07H
つぎに一致するファイル名の検索	Find Next File	4FH
つぎのエントリを検索	Search For Next Entry	12H
ディスク転送アドレスを得る	Get Disk Transfer Address	2FH
ディスク転送アドレスのセット	Set Disk Transfer Address	1AH
ディスクの選択	Select Disk	0EH
ディスクのフリースペースを得る	Get Disk Free Spece	36H
ディレクトリエントリの削除	Delete Directory Entry	41H
ディレクトリエントリの変更	Change Directory Entry	56H
ディレクトリの作成	Create Directory	39H
ディレクトリの変更	Remove Directory	3AH
デフォルトドライブのデータを得る	Get Default Drive Data	1BH
ドライブのデータを得る	Get Drive Data	1CH
ハンドルのオープン	Open Handle	3DH
ハンドルのクローズ	Close Handle	3EH
ハンドルの作成	Create Handle	3CH
バッファを空にしてキーボード入力	Flush Buffer, Read Keyboard	0CH
バッファードキーボード入力	Buffered Keyboard Input	0AH
日付を得る	Get Date	2AH
日付をセットする	Set Date	2BH
PSP を得る	Get PSP	62H
ファイルアクセスのロック	Lock	5C00H
ファイルアクセスのロック解除	Unlock	5C01H
ファイルのオープン	Open File	0FH
ファイルの大きさを得る	Get File Size	23H
ファイルのクローズ	Close File	10H
ファイルの削除	Delete File	13H

ファイルの作成	Create File	16H
ファイルの日付/時刻を得る/セットする	Get/Set Date/Time of File	57H
ファイルハンドルの強制二重化	Force Duplicate File Handle	46H
ファイルハンドルの二重化	Duplicate File Handle	45H
ファイルポインタの移動	Move File Pointer	42H
ファイル名の解析	Parse File Name	29H
ファイル名の変更	Rename File	17H
プリンタセットアップ	Printer Setup	5E02H
プログラムの終了	Terminate Program	00H
プログラムのロードと実行	Load and Execute Program	4B00H
プロセスの終了	End Process	4CH
ペリファイの状態を得る	Get Verify State	54H
ペリファイフラグのセット/リセット	Set/Reset Verify Flag	2EH
補助出力	Auxiliary Output	04H
補助入力	Auxiliary Input	03H
マシン名を得る	Get Machine Name	5E00H
メモリの割り当て	Allocate Memory	48H
文字のスクリーン出力	Display Character	02H
文字のプリンタ出力	Print Character	05H
ライトハンドル	Write Handle	40H
ランダムブロックライト	Random Block Write	28H
ランダムブロックリード	Random Block Read	27H
ランダムライト	Random Write	22H
ランダムリード	Random Read	21H
リードハンドル	Read Handle	3FH
リセット ディスク	Reset Disk	0DH
論理ドライブマップの取得	Get Logical Drive Map	440EH
論理ドライブマップの設定	Set Logical Drive Map	440FH
割り当てられたメモリの解放	Free Allocated Memory	49H
割り当てられたメモリブロックの変更	Set Block	4AH
割り当てリストのエントリを得る	Get Assign List Entry	5F02H
割り当てリストのエントリのキャンセル	Cancel Assign List Entry	5F04H
割り当てリストのエントリの作成	Make Assign List Entry	5F03H
割り込みベクタを得る	Get Interrupt Vector	35H
割り込みベクタのセット	Set Interrupt Vector	25H

INT 21H

Terminate Program

ファンクション 00H

機能 プログラムの終了

コール AH = 00H

CS プログラムセグメントプレフィックスのセグメントアドレス

リターン なし

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 ファンクション 00H は割り込みタイプ 20H をコールし、同じ処理を行います。この割り込みを実行するためには、その前に CS レジスタにプログラムセグメントプレフィックスのセグメントアドレスを入れておかなければなりません。以下の抜け出しアドレスは、プログラムセグメントプレフィックス内の指定されたオフセットアドレスから回復されます。

抜け出しアドレス	オフセット
プログラムの終了	0AH
CTRL-C	0EH
重大なエラー	12H

ファイルバッファは、すべてディスクに書き出されます。

警告： このファンクションコールを行うためには、その前に大きさを変更したすべてのファイルをクローズしておかなければなりません。

変更されたファイルがクローズされていない場合、ファイルの大きさはディレクトリに正しく記録されません。ファイルクローズシステムコールについては、ファンクション 10H を参照してください。


```
マクロ定義  terminate_program macro
                xor         ah, ah
                int         21H
                endm
```

例 メッセージを出力して、MS-DOSに戻るプログラムをつぎに示します。ただし、これは 1.9.6 のサンプルプログラムのようなプログラムのサブルーチンとして使われます。

```
message db "Displayed by FUNC00H example", 0DH, 0AH, "$"
func_00H: display message      ; メッセージ表示(09H)
          terminate_program    ; プログラムの終了
code      ends
          end      start
```

INT 21H

Read Keyboard And Echo

ファンクション 01H

機能 キーボード入力とエコー

コール AH = 01H

リターン AL 入力された文字

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 標準入力（キーボード）から1文字入力されるまで待ち、入力された文字を標準出力（スクリーン）に出力し、そのキャラクタコードをALレジスタに返します。この入力が〈CTRL-C〉の場合、割り込みタイプ23Hを実行します。

マクロ定義

```
read_kbd_and_echo macro
    mov     ah, 01H
    int     21H
endm
```

例 つぎのプログラムは、文字を入力したとおりにスクリーンとプリンタに出力するものです。リターンキーが押されると改行キャリッジリターン（コード）が両方に出力されます。

```
func_01H: read_kbd_and_echo      ; キーボード入力とエコー
          print_char    al        ; プリンタに出力(05H)
          cmp           al, 0DH    ; キャリッジリターンコードか?
          jne           func_01H   ; いいえの時、次の文字の入力待ち
          print_char     10        ; 改行コードをプリンタに出力(05H)
          display_char   10        ; 改行コードをスクリーンに出力(02H)
          jmp           func_01H   ; 他の文字の入力待ち
```

INT 21H

Display Character

ファンクション 02H

機能	文字のスクリーン出力
コール	AH = 02H DL スクリーン出力すべき文字
リターン	なし

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 DL 内の文字を標準出力（スクリーン）に出力します。〈CTRL-C〉が入力された場合は、割り込みタイプ 23H が実行されます。

マクロ定義

```

display-char macro character
    mov     dl, character
    mov     ah, 02H
    int     21H
endm

```

例 つぎのプログラムは、小文字を大文字に変換してスクリーンに表示するものです。

```

func_02H:  read_kbd          ; キーボード入力(08H)
            cmp     al, "a"
            jl      uppercase ; 変換しない(英小文字でない)
            cmp     al, "z"
            jg      uppercase ; 変換しない(英小文字でない)
            sub     al, 20H    ; 大文字の ASCII コードに変換
uppercase:  display_char al   ; 大文字をスクリーンに出力
            jmp     func_02H   ; 他の文字の入力待ち

```


INT 21H

Auxiliary Input

ファンクション 03H

機能	補助入力
コール	AH = 03H
リターン	AL 補助装置から入力された文字

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

- 解説** 補助入力装置から1文字入力されるまで待ち、入力された文字コードをALに返します。このシステムコールは、ステータスやエラーコードを返しません。
 <CTRL-C>が入力されると割り込みタイプ23Hが実行されます。

マクロ定義

```

aux_input macro
    mov     ah, 03H
    int     21H
endm

```

- 例** つぎのプログラムは、補助装置から入力された文字をそのとおりにプリンタ出力するものです。エンドオブファイルコード (ASCII コード 1AH, <CTRL-Z>) が入力されると、出力を停止します。

```

func_03H:  aux_input          ; 補助入力装置からの入力
            cmp     al, 1AH    ; ファイルエンドか?
            je      continue   ; はいの時、出力を停止
            print_char al      ; 入力文字をプリンタに出力(05H)
            jmp     func_03H    ; 他の文字の入力待ち

continue:  .
            .

```

INT 21H

Auxiliary Output

ファンクション 04H

機能 補助出力

コール AH = 04H
DL 補助装置に出力すべき文字

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DL 内の文字を，補助出力装置に出力します。このシステムコールは，ステータスやエラーコードを返しません。

〈CTRL-C〉が入力されると割り込みタイプ 23H が実行されます。

```
マクロ定義  aux_output macro character
               mov     dl, character
               mov     ah, 04H
               int     21H
               endm
```

例 つぎのプログラムは，キーボードから入力された最高 80 バイトまでの一連のストリングス（文字列）を補助装置に出力するものです。このプログラムは，ヌルストリング（CR のみ）が入力されると停止します。

```
string db 81 dup (?) ; ファンクション 0AH 参照
;
func_04H: get_string 80, string ; キーボードから最大 80 バイト
; 入力する (0AH)
               cmp     string [1], 0 ; ヌルストリングか?
               je      continue ; はいの時, 停止
```

```
      mov      cx, word ptr string [1]    ; スtring長を得る
      mov      bx, 0                      ; インデックス(BX)に0をセット
send_it: aux_output string [bx + 2]       ; 補助装置に出力
      inc      bx                         ; インデックスをインクリメント
      loop     send_it                    ; 次の文字出力処理
      jmp      func_04H                   ; 他のStringの入力/出力処理へ
```

```
continue:  •
           •
```


INT 21H

Print Character

ファンクション 05H

機能	文字のプリンタ出力
コール	AH = 05H DL プリンタに出力すべき文字
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DL 内の文字をプリンタに出力します。〈CTRL-C〉が入力されると、割り込みタイプ 23H が実行されます。このファンクションはステータスやエラーコードを返しません。

マクロ定義

```

print_char macro character
    mov     dl, character
    mov     ah, 05H
    int     21H
endm

```

例 つぎのプログラムは、プリンタ内にテストパターンを出力するものです。このプログラムは、**CTRL**+**C** が押されると停止します。

```

line_num    db      0
;
func_05H:   mov     cx, 60      ; プリント出力ライン数を 60 とする
start_line: mov     bl, 33      ; 最初にプリント可能な ASCII
                                ; 文字は(!)である
                                ;
                                add     bl, line_num ; 出力する文字のオフセットをセット
                                push    cx         ; プリント出力ラインカウンタをセーブ
                                mov     cx, 80      ; 1 行文の文字数(80)を CX にセット
print_it:   print_char bl       ; プリントに文字を出力
                                inc     bl         ; 次の ASCII 文字の出力準備
                                cmp     bl, 126    ; 出力可能な最後の
                                ; ASCII 文字( )か?
                                jl      no_reset   ; まだのときは no_reset へ
                                mov     bl, 33     ; 文字(!)から始める
no_reset:   loop     print_it    ; 次の文字の出力処理へ
                                print_char 13      ; キャリッジリターンコードの出力
                                print_char 10      ; ラインフィードコードの出力
                                inc     line_num    ; オフセットをインクリメント
                                pop     cx         ; プリント出力ラインカウンタをリストア
                                loop     start_line ; 次のラインをプリント

```

INT 21H

Direct Console I/O

ファンクション 06H

機能 直接コンソール I/O

コール AH = 06H
DL 解説の項を参照。

リターン AL

コールする前に、DL = FFH の場合：
ゼロフラグがセットされていないならば、AL にキ
ャラクタが入ります。
ゼロフラグがセットされていれば、キャラクタ
は無く、AL = 00H になります。
コールする前に、DL ≠ FFH の場合：
なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 処理は、このファンクションコールが行われたときの DL 内の値により変わります。

DL = FFH 標準入力（キーボード）から文字が入力された場合、この文字が AL 内に返され、ゼロフラグが 0 になります。文字が入力されていない場合、ゼロフラグは 1 になります。

DL ≠ FFH DL 内の文字が、標準出力（スクリーン）に出力されます。

この機能は、〈CTRL-C〉の検査を行いません。

マクロ定義 dir_console_io macro switch
mov dl, switch
mov ah, 06H
int 21H
endm

例

つぎのプログラムは、システムクロックを0にセットし、時刻を継続的にスクリーンに表示するものです。いずれかの文字が入力されたとき、時刻の表示が停止します。再びいずれかの文字が入力されると、このクロックは0にリセットされ、時刻の表示が再開します。

```

time          db "00 : 00 : 00,00",13,10, "$" ;"$"の説明は
                                                    ;ファンクション 09H 参照

ten           db 10
;

func_06H:     set_time      0, 0, 0, 0          ;時刻をセット(2DH)
read_clock :   get_time      ;時刻を得る(2CH)
               convert      ch, ten, time       ;章末参照
               convert      cl, ten, time [3]    ;章末参照
               convert      dh, ten, time [6]    ;章末参照
               convert      dl, ten, time [9]    ;章末参照
               display      time                 ;time をスクリーンに出力(09H)
               dir_console_io OFFH              ;任意の文字を入力
               jne          stop                 ;入力ありの時、時刻表示の停止
               jmp          read_clock           ;入力なしの時、時刻表示の継続
                                                    ; running

stop:         read_kbd      ;キーボード入力待ち(08H)
               jmp         func_06H             ;時刻表示を再開

```

INT 21H

Direct Console Input

ファンクション 07H

機能	直接コンソール入力
コール	AH = 07H
リターン	AL キーボードから入力された文字

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説	標準入力(キーボード)から文字が入力されるまで待ち、つぎにこの文字を AL に返します。このファンクションは、文字のエコーや<CTRL-C>の検査は行いません。エコーまたは<CTRL-C>の検査を行うキーボード入力ファンクションについては、ファンクション 01 H または 08 H を参照してください。
----	--

マクロ定義	<pre> dir-console_input macro mov ah, 07H int 21H endm </pre>
-------	---

例	つぎのプログラムは、8 文字までのパスワード入力を促すプロンプトを表示し、エコーを行わないでこの文字をストリング内に入れるものです。
---	--

```

password db 8 dup (?)
prompt   db "password: $ "      ;"$"の説明は
                                   ;ファンクション 09H 参照

```

```

func_07H: display prompt          ; prompt をスクリーンに出力(09H)
        mov     cx, 8              ; 入力可能なパスワードの最大値 8 を
                                   ; セット
        xor     bx, bx             ; BX はパスワードのインデックス
                                   ; として使用
get_pass: dir_console_input        ; キーボードから入力された文字を AL
                                   ; に返す
        cmp     al, 0DH            ; キャリッジリターンか?
        je      continue          ; はいの時、処理終了
        mov     password[bx], al   ; いいえの時、この文字をstring内
                                   ; に入れる
        inc     bx                 ; インデックスをインクリメント
        loop    get_pass           ; 次の文字を得る
continue:  .                       ; BX はパスワード+1 の長さである
        .

```


INT 21H

Read Keyboard

ファンクション 08H

機能 キーボード入力

コール AH = 08H

リターン AL キーボードから入力された文字

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 標準入力(キーボード)から1文字入力されるまで待ち、つぎにこの文字をALに返します。〈CTRL-C〉が入力された場合、割り込みタイプ23Hが実行されます。このファンクションは、文字のエコーを行いません(文字のエコーや〈CTRL-C〉の検査を行うキーボード入力ファンクションについては、ファンクション01Hを参照してください)。

マクロ定義

```
read_kbd macro
    mov     ah, 08H
    int     21H
endm
```

例 つぎのプログラムは、8文字までのパスワードの入力を促すためプロンプトを表示し、エコーを行わずに文字をストリング内に入れるものです。

```
password db 8 dup (?)
prompt   db "password: $"           ; "$"の説明は
                                         ; ファンクション09H参照
```

```

func_08H:  display prompt           ; prompt をスクリーンに出力(09H)
            mov     cx, 8             ; 入力可能なパスワードの最大値 8 をセット
            xor     bx, bx           ; BX はパスワードのインデックスとして
            ; 使用

get_pass:   read_kbd                ; キーボードから入力された文字を
            ; AL に返す

            cmp     al, 0DH          ; キャリッジリターンか?
            je      continue         ; はいの時, 処理終了
            mov     password[bx], al ; いいえの時, この文字をstring内に
            ; 入れる
            inc     bx               ; インデックスをインクリメント
            loop    get_pass         ; 次の文字を得る

continue:   .
            .
            .

```

INT 21H

Display String

ファンクション 09H

機能 スtringのスクリーン出力

コール AH = 09H

DS : DX スクリーンに出力すべき文字列の
 先頭アドレス

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSh	FLAGSL

CS
DS
SS
ES

解説 DX には、“\$” で終わるStringのオフセット (DS にはセグメントアドレス) が入っていなければなりません。このStringが、標準出力 (スクリーン) に出力されます (\$ は、スクリーンに出力されません)。

マクロ定義

```
display macro string
        mov     dx, offset string
        mov     ah, 09H
        int     21H
        endm
```

例 つぎのプログラムは、入力されたキーの16進コードをスクリーンに出力するものです。

```
table      db      "0123456789ABCDEF"
sixteen    db      16
result     db      "- 00H", 13, 10, "$"      ; $の説明は
                                              ; 本文を参照
```



```
func_09H: read_kbd_and_echo          ; キーボード入力された文字
                                         ; をスクリーンに出力(01H)

        convert    al, sixteen, result [ 3 ]    ; 章末参照
        display    result                    ; 入力されたキーの16進コ
                                         ; ードをスクリーンに出力

        jmp        func_09H                ; 処理継続
```

INT 21H

Buffered Keyboard Input

ファンクション 0AH

機能	バッファードキーボード入力
コール	AH = 0AH DS : DX 入力バッファへのポインタ
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 DX にはバッファのオフセット (DS にはセグメントアドレス)が入っていないければなりません。ストリングは標準入力 (キーボード) から入力されます。また、入力バッファはつぎのようなフォーマットをしています。

オフセット	内容
1	CR (キャリッジリターンコード) まで含めたバッファ内の最大文字数 (ユーザーが、セットします。)
2	実際に入力された、CR を入れない文字数 (この値は、このファンクションによってセットされます。)
3~n	バッファ領域 (バイト 1 で指定した大きさ以上でなければなりません。)

このファンクションは、文字が入力されるまで待ち、文字がキーボードから入力されると、リターンキーが押されるまで、3 バイト目以後のバッファに入れる処理が続きます。バッファが最後のバイトの 1 つ前まで埋められると、それ以後に入力された文字は無視され、リターンキーが押されるまで ASCII コード 7 (BEL) がターミナルに出力されます。このストリングは、編集することができます。

〈CTRL-C〉が入力されると、割り込みタイプ 23H が実行されます。

MS-DOS はこのバッファの 2 バイト目を、入力された文字数(CR を入れない)にセットします。

マクロ定義

```
get_string macro limit, string
    mov     dx, offset string
    mov     string, limit
    mov     ah, 0AH
    int     21H
endm
```

例

つぎのプログラムは、キーボードから最大 16 バイトまでのストリングを入力し、24 行×80 文字のスクリーンをこのストリングで埋めるものです。

```
buffer          label    byte
max_length      db       ?           ; 最大長
chars_entered   db       ?           ; 文字数
string          db       17dup (?)   ; 16 文字+CR
strings_per_line dw      0           ; 1 行に出力可能なストリング数
crlf            db       13, 10, "$"
;
func_0AH:       get_string 17, buffer ; バッファードキーボード入力
                xor      bx, bx      ; BX はバッファのインデックスとして
                ; バイト単位で使用

                mov      bl, chars_entered ; ストリング長を得る
                mov      buffer [bx+2], "$" ; "$" のセット (09H)
                mov      al, 50H      ; ラインあたりのカラム数を指定
                cbw
                div      chars_entered ; 1 行あたりのストリング数を算出
                xor      ah, ah       ; 残りをクリア
                mov      strings_per_line, ax ; カラムカウンタをセーブ
                mov      cx, 24      ; ラインカウンタをセット
display_screen: push    cx           ; それをセーブ
                mov      cx, strings_per_line ; カラムカウンタを得る
display_line:   display string       ; string をスクリーンに出力 (09H)
                loop     display_line
                display   crlf        ; CRLF をスクリーンに出力 (09H)
                pop      cx          ; ラインカウンタを得る
                loop     display_screen ; 次の 1 行表示へ
```


INT 21H

Check Keyboard Status

ファンクション 0BH

機能 キーボードステータスの検査

コール AH=0BH

リターン AL= FFH タイプaheadバッファ内に文字が入っている。
 00H タイプaheadバッファ内に文字が入っていない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 標準入力（標準入力がない場合はタイプaheadバッファ内）に文字が入っているかどうかを検査します。入っている場合、AL に FFH (255) が、入っていない場合、00H が返されます。〈CTRL-C〉がバッファ内に入っている場合、割り込みタイプ 23H が実行されます。

マクロ定義 check_kbd_status macro

```

mov    ah, 0BH
int     21H
endm

```

例 つぎのプログラムは、いずれかのキーが押されるまで、時刻を継続的にスクリーンに出力するものです。

```
time    db    "00 : 00 : 00. 00", 13, 10, "$"
```

```
ten     db    10
```

```

.
.

```

```
func_0BH: get_time ;時刻を得る(2CH)
```

```

convert  ch, ten, time      ; 章末参照
convert  cl, ten, time [3]  ; 章末参照
convert  dh, ten, time [6]  ; 章末参照
convert  dl, ten, time [9]  ; 章末参照
display  time               ; time をスクリーンに出力(09H)
check_kbd_status           ; キーボードステータスの検査
cmp      al, 0FFH          ; タイプaheadバッファ内に文字が
                           ; 入っているか?

je       all_done          ; はいの時, 処理終了
jmp      func_0BH          ; いいえの時, 時刻を継続的に
                           ; スクリーン出力

```

```
all_done:  •
```

```
          •
```

```
          •
```

INT 21H

Flush Buffer, Read Keyboard

ファンクション 0CH

機能 バッファを空にして、キーボード入力

コール AH = 0CH

AL = 01H, 06H, 07H, 08H, 0AH :

対応するファンクションのコールが行われる。

他の値：これ以上の処理は行われない。

リターン AL=00H タイプaheadバッファは、空になっている。これ以上の処理は行われない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 標準入力バッファ（標準入力がない場合はタイプaheadバッファ）を空にします。これ以上の処理を行うかどうかは、このファンクションコールが行われた時の AL 内の値によります。

01H, 06H, 07H, 08H, 0AH……対応する MS-DOS ファンクションが、実行されます。

他の値……これ以上の処理は行われず、AL に 0 が返されます。

マクロ定義

```
flush-and-read-kbd macro switch
    mov     al, switch
    mov     ah, 0CH
    int     21H
endm
```

例 つぎのプログラムは、文字を入力したとおりにスクリーンとプリンタに出力するものです。リターンキーが押された時、このプログラムによってキャリッジリターン改行（コード）が両方に出力されます。

```
func_0CH: flush-and-read-kbd 1      ; バッファを空にして、キーボード入力
          print-char    al          ; 入力された文字をプリンタに出力(05H)
```



```
cmp      al, 0DH      ; キャリッジリターンか?
jne      func_0CH      ; いいえの時, プリンタに出力
print_char 10          ; はいの時, プリンタに改行コードを
                        ; 出力(05H)
display_char 10        ; スクリーンに改行コードを出力(02H)
jmp      func_0CH      ; 次の文字を得る
```

INT 21H

Reset Disk

ファンクション 0DH

機能 リセットディスク

コール AH = 0DH

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 内部バッファ記憶域がドライブ内のディスクと一致しているかどうか確認するために使用します。このファンクションによって、修正が行われたバッファが書き出され、内部記憶域にあるすべてのバッファに解放されているという印が付けられます。

このファンクション 0DH はすべてのファイルバッファを書き出します。ディレクトリエントリの更新は行わないので、ユーザは、ディレクトリエントリの更新を行うために変更されたファイルをクローズしなければなりません（ファンクション 10H のクローズファイルを、参照してください）。

マクロ定義

```

reset_disk macro disk
            mov     ah, 0DH
            int     21H
        endm

```

例 reset_disk ; このコールにはエラーリターンはない

INT 21H

Select Disk

ファンクション 0EH

機能 ディスクの選択

コール AH = 0EH

DL ドライブ番号 (00H = A:, 01H = B:, など)

リターン AL 論理ドライブの数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DL で指定されたドライブ (00H=A:, 01H=B: …) が、カレントのディスクとして選択されます。AL にドライブ数が返されます。

注意:

将来の互換性のために、AL に返された値は注意深く扱ってください。AL に返される論理ドライブ数は、実際に接続されているドライブ数とは限らず、CONFIG.SYS の LASTDRIVE 指定によって変わります。AL の最小値は 5 (LASTDRIVE 指定なしで実際のドライブが 5 台以下のとき) ですので、AL が 05 H を返したからといって A, B, C, D, E の各ドライブがすべて有効なドライブ指定とは限りません。

```
マクロ定義 select_disk macro disk
                mov     dl, disk - "A"
                mov     ah, 0EH
                int      21H
                endm
```


例

つぎのプログラムは、2ドライブシステムで現在選択されていないドライブを
カレントディスクにするものです。

```
func_0EH:  current_disk      ; カレントディスクを得る(19H)
            cmp      al, 00H  ; ドライブ A が選択されているか?
            je       select_b ; はいの時, select_b へ
            select_disk "A"   ; いいえの時, ドライブ A を選択
            jmp      continue
select_b:   select_disk "B"   ; ドライブ B を選択
continue:   .
            .
```

INT 21H

Open File

ファンクション 0FH

機能 ファイルのオープン

コール AH = 0FH
DS : DX オープンされていない FCB

リターン AL = 00H ディレクトリエントリが存在する。
FFH ディレクトリエントリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされていないファイルコントロールブロック (FCB) のオフセット (DS 内のセグメントアドレスから) が入っていなければなりません。指定された名前のファイルを見つけるために、ディスクディレクトリを検索します。このファイルのディレクトリエントリが存在する場合、AL に 00H が返され、FCB は以下のようにセットされます。

- ドライブコードが 00H (カレントディスク) の場合、実際に使用されているディスク番号 (01H=A:, 02H=B:, ...) に変更されます。これにより、このファイルで引き続き行われる動作を妨害することなく、カレントディスクを変更することができます。
- 現在のブロックフィールド (オフセット 0CH) は、ゼロにセットされます。
- レコードサイズ (オフセット 0FH) は、システムデフォルト値である 128 にセットされます。
- ファイルのサイズ (オフセット 10H)、最後に書き込みが行われた日付 (オフセット 14H) と時刻 (オフセット 16H) が、ディレクトリエントリから得られた情報を使用してセットされます。

このファイルに対してシーケンシャルなディスクアクセスを行う場合は、事前

に現在のレコードフィールド(オフセット 20H)を、ランダムなディスクアクセスを行う場合は、相対レコードフィールド(オフセット 21H)をセットしておかなければなりません。デフォルトレコードサイズ(128 バイト)を使用しない場合は、正しい長さにセットしてください。

ファイルのディレクトリエントリが存在しないか、属性がシステムあるいは隠されたファイルの場合、AL に FFH (255) が返されます。

マクロ定義

```
open    macro fcb
        mov    dx, offset fcb
        mov    ah, 0FH
        int    21H
        endm
```

例

つぎのプログラムは、ドライブ B: に存在するディスク上の TEXTFILE.ASC という名前のファイルをプリンタに出力するものです。バッファの中のレコードにエンドオブファイルコード(ASCII コード 1AH, <CTRL-Z>)が含まれているときは、それが検出されるまで文字が出力されます。

```
fcb      db      2, "TEXTFILEASC"
          db      25 dup (?)
buffer   db      128 dup (?)
;
func_0FH: set_dta  buffer          ; ディスク転送アドレスのセット(1AH)
          open    fcb              ; TEXTFILE.ASC ファイルのオープン
read_line: read_seq fcb            ; シーケンシャルリード(14H)
          cmp     al, 01H          ; ファイルエンドか?
          je      all_done         ; はいの時, all_done へ
          cmp     al, 00H          ; ディレクトリエントリが存在するか?
          jg      check_more       ; いいえの時, check_more へ
          ; record
          mov     cx, 128          ; はいの時, バッファ中のレコードを
          ; プリンタに出力
          xor     si, si           ; インデックスを 0 にセット
print_it  print_char buffer [si]   ; バッファ中の文字をプリンタに出力(05H)
          inc     si              ; インデックスをインクリメント
          loop    print_it         ; 次の文字をプリンタに出力
```



```

                                jmp      read_line      ; 次のレコードをリード
check_more:                   cmp      al, 03H          ; プリントするレコードがあるか?
                                jne      all_done        ; いいえの時, all_done へ
                                mov      cx, 128         ; はいの時, バッファ中のレコードを
                                                ; プリンタに出力
                                xor      si, si          ; インデックスを0にセット
find_eof:                     cmp      buffer [si], 26   ; ファイルエンドか?
                                je       all_done        ; はいの時, all_done へ
                                print_char buffer [si]   ; バッファ中の文字をプリンタに出力
                                                ; (05H)
                                inc      si             ; インデックスをインクリメントする
                                loop     find_eof
all_done:                     close    fcb              ; ファイルのクローズ(10H)

```

INT 21H

Close File

ファンクション 10H

機能 ファイルのクローズ

コール AH = 10H
DS : DX オープンされている FCB

リターン AL = 00H ディレクトリエントリが存在する,
FFH ディレクトリエントリが存在しない,

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされている FCB のオフセット (DS にはセグメントアドレス)が入っていなければなりません。FCB で指定されたファイルを見つけるために、ディスクディレクトリの検索が行われます。ファイルが変更された場合、このファンクションコールを行わなければディレクトリエントリは更新されません。

このファイルのディレクトリエントリが存在する場合、ファイルのロケーションが FCB 内の対応するエントリと比較されます。必要に応じて、FCB と一致させるため、エントリを更新し、AL に 00H が返されます。

ファイルのディレクトリエントリが存在しない場合、AL に FFH (255) が返されます。

マクロ定義

```
close macro fcb
    mov     dx, offset fcb
    mov     ah, 10H
    int     21H
endm
```

例

つぎのプログラムは、ドライブ B: に存在する MOD1.BAS という名前のファイルの先頭のバイトが FFH かどうか調べ、FFH の場合メッセージをプリンタに出力するものです。

```

message    db    "Not saved in ASCII format", 13, 10, "$"
fcb        db    2, "MOD1  BAS"
           db    25 dup (?)
buffer     db    128 dup (?)
;
func-10H:  set_dta  buffer           ; ディスク転送アドレスのセット(1AH)
           open    fcb              ; MOD1.BAS ファイルのオープン(0FH)
           read_seq fcb             ; シーケンシャルリード(14H)
           cmp     buffer, 0FFH     ; ファイルの先頭バイトは FFH か?
           jne     all_done         ; いいえの時, all_done へ
           display message         ; はいの時, message をプリンタへ出力(09H)
all_done:  close   fcb              ; ファイルのクローズ

```


INT 21H

Search For First Entry

ファンクション 11H

機能 最初のエントリを検索

コール AH=11H

DS : DX オープンされていない FCB

リターン AL= 00H ディレクトリエントリが存在する。
 FFH ディレクトリエントリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSh	FLAGSl

CS
DS
SS
ES

解説

DX には、オープンされていない FCB のオフセット (DS にはセグメントアドレス)が入っていなければなりません。最初に一致する名前を見つけるために、ディスクディレクトリを検索します。ファイル名にはワイルドカード文字を使用することができます。隠れて見えないファイルやシステムファイルを検索する場合、DX は拡張 FCB の先頭のバイトを示していなければなりません。

FCB 内のファイル名のディレクトリエントリが存在する場合、AL に 0 が返され、同じ種類 (通常または拡張) のオープンされていない FCB がディスク転送アドレスに作成されます。

FCB 内のファイル名のディレクトリエントリが存在しない場合、AL に FFH (255) が返されます。

検索している FCB が通常の FCB の場合、ディスク転送アドレスの最初の 1 バイトには、使われているドライブ番号がセットされ、つぎの 32 バイトがディレクトリエントリです。

検索している FCB が拡張 FCB の場合、ディスク転送アドレスの最初の 1 バイトには FFH が、つぎの 5 バイトには 00 H がセットされ、それに続く 1 バイトに検索しているファイルの属性が示されています。残りの 33 バイトは通常の FCB の場合と同じです (1 バイトのドライブ番号と 32 バイトのディレクトリエントリ)。

ファンクション 12H (つぎのエントリを検索) を使ってファイル名を検索する場合、DS:DX にあったもとの FCB は決してオープンも変更もされてはなりません。

注意

アトリビュート (属性) フィールドは、拡張 FCB の最後のバイトで、FCB の前に位置します (拡張 FCB の詳細は、"1.8.3 拡張 FCB" を参照)。

拡張 FCB が用いられた場合は、つぎのような検索が行われます。

1. FCB のアトリビュート (属性) がゼロの場合、通常のファイルエントリだけが検索される。ボリュームラベル、サブディレクトリ、隠されたシステムファイルは検索されない。
2. アトリビュート (属性) フィールドが、隠れたファイル、システムファイル、ディレクトリエントリ (02 H, 04 H, 10 H) またはその任意の組み合わせにセットされた場合、通常のファイルの他に、これらも検索されるようになる。これはアトリビュートバイトが 16H (隠された+システム+ディレクトリ (3 ビットすべてが ON)) にセットされた場合で、ボリュームラベルだけは除外される。
3. アトリビュートフィールドがボリュームラベル (08 H) にセットされた場合、ボリュームラベルエントリだけが検索され、他は対象から除外される。

マクロ定義

```
search_first macro fcb
    mov     dx, offset fcb
    mov     ah, 11H
    int     21H
endm
```

例

つぎのプログラムは、ドライブ B: に REPORT.ASM という名前のファイルが存在するかどうかを検索するものです。

```
yes      db    "FILE EXISTS. $"
no       db    "FILE DOES NOT EXIST. $"
fcb      db    2, "REPORT ASM"
         db    25 dup (?)
buffer   db    128 dup (?)
;
func_11H: set_dta    buffer      ; ディスク転送アドレスのセット (1AH)
         search_first fcb        ; REPORT.ASM ファイルの検索
```

```

        cmp     al, 0FFH    ; ディレクトリエントリが存在するか?
        je      not_there   ; いいえの時, not_there へ
        display yes         ; はいの時, yes をスクリーンに出力(0AH)
        jmp     continue
not_there: display no       ; no をスクリーンに出力(09H)
continue: display crlf      ; CRLF をスクリーンに出力(09H)
    
```


INT 21H

Search For Next Entry

ファンクション 12H

機 能	つぎのエントリを検索
コール	AH = 12H DS : DX オープンされていない FCB
リターン	AL = 00H ディレクトリエントリが存在する。 FFH ディレクトリエントリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解 説 DX には、前にファンクション 11H のコールを行ったときに指定された FCB のオフセット (DS 内のセグメントアドレスから) が入っていなければなりません。このファンクションは、ファイル名にワイルドカード文字が使われたとき、他のディレクトリエントリを見つけるために、ファンクション 11H (最初のエントリ検索) の後使用します。つぎに一致する名前を見つけるためにこのディスクディレクトリが、検索されます。ファイル名にはワイルドカード文字を使用することができます。隠れて見えないファイルまたはシステムファイルを検索する場合、DX は拡張 FCB の先頭のバイトを示していなければなりません。

FCB 内のファイル名のディレクトリエントリが存在する場合、AL に 00H が返され、同じ種類 (通常または拡張) のオープンされていない FCB がディスク転送アドレスに作成されます。

FCB 内のファイル名のディレクトリエントリが存在しない場合、AL に FFH (255) が返されます (オープンされていない FCB についてはファンクション 11H を参照してください)。

```

マクロ定義      search_next macro fcb
                    mov     dx, offset fcb
                    mov     ah, 12H
                    int      21H
                    endm

```

例 つぎのプログラムは、ドライブ B: に存在するファイル数をスクリーンに出力するものです。

```

message      db  "No files", 10, 13, "$"
files         db  0
ten           db  10
fcb           db  2, "?????????????"
              db  25 dup (?)
buffer        db  128 dup (?)
;
func_12H:     set_dta  buffer          ; ディスク転送アドレスのセット(1AH)
              search_first fcb         ; 最初のエントリを検索(11H)
              cmp      al, 0FFH        ; ディレクトリエントリが存在するか?
              je        all_done        ; いいえの時, all_done へ
              inc       files           ; はいの時, ファイル数に1を加える
              ; counter
search_dir:   search_next fcb          ; 次のエントリを検索
              cmp      al, 0FFH        ; ディレクトリのエントリが存在するか?
              je        done            ; はいの時, ファイル数に1を加える
              inc       files           ; counter
              jmp       search_dir      ; 再チェックする
done:         convert  files, ten, message ; 章末参照
all_done:     display  message          ; message をスクリーンに出力(09H)

```

INT 21H

Delete File

ファンクション 13H

機能 ファイルの削除

コール AH = 13H
DS : DX オープンされていない FCB

リターン AL = 00H ディレクトリエントリが存在する。
FFH ディレクトリエントリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされていない FCB のオフセット (DS にはセグメントアドレス) が入っていなければなりません。目的のファイル名を見つけるために、ディレクトリが検索されます。FCB 内のファイル名には、ワイルドカード文字を用いることができます。

一致するディレクトリエントリが存在する場合、このエントリはディレクトリから削除され AL に 00H が返されます。このファイル名にワイルドカード文字が使用されている場合、該当するすべてのディレクトリエントリが削除されます。

一致するディレクトリエントリが存在しない場合、AL に FFH (255) が返されます。

マクロ定義

```
delete macro fcb
    mov     dx, offset fcb
    mov     ah, 13H
    int     21H
endm
```

例 つぎのプログラムは、ドライブ B: に存在するファイルのうち、1982 年 12 月 31 日以前に編集されたものを削除するものです。


```

year      dw      1982
month     db      12
day       db      31
files     db      0
ten       db      10
message   db      "NO FILES DELETED.", 13, 10, "$"
; $の説明はファンクション 09H を参照

fcb       db      2, "?????????????"
          db      25 dup (?)
buffer    db      128 dup (?)
;
func_13H: set_dta buffer          ; ディスク転送アドレスのセット(1AH)
          search_first fcb        ; 最初のエントリの検索(11H)
          cmp          al, 0FFH    ; ディレクトリエントリは存在するか?
          je           all_done    ; いいえの時, all_done へ
compare:  convert_date buffer     ; 章末参照
          cmp          cx, year    ; CX(年)DL(月)DH(日)を
          jg           next        ; それぞれ year, month, day と
          cmp          dh, month   ; 比較する
          jg           next        ; 1982 年 12 月 31 日以前ならば
          cmp          dh, day     ; ファイルを削除
          jge          next
          delete       buffer      ; ファイルの削除
          inc          files       ; 削除ファイルカウンタを
          ; インクリメントする
next:     search_next fcb         ; 次のエントリを検索(12H)
          cmp          al, 00H     ; ディレクトリエントリは存在するか?
          je           compare     ; はいの時, 日付のチェック
          cmp          files, 0    ; いくつかファイルを削除したか?
          je           all_done    ; いいえの時 NO FILES メッセージを表示
          convert      files, ten, message ; 章末参照
all_done: display      message     ; message をスクリーンに出力(09H)

```

INT 21H

Sequential Read

ファンクション 14H

機能 シーケンシャルリード

コール AH = 14H
DS : DX オープンされている FCB

リターン AL = 00H 正常な読み込み
01H EOF
02H ディスク転送アドレス (DTA)
で示されるバッファが小さすぎる。
03H EOF, レコードの一部

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされている FCB のオフセット (DS にはセグメントアドレス) が入っていなければなりません。カレント (現在の) ブロック (オフセット 0CH) およびカレント (現在の) レコード (オフセット 20H) フィールドによって示されているレコードが、ディスク転送アドレスにロードされ、次にカレントのブロックおよびカレントのレコードフィールドがつぎのレコードを示すようにセットされます。

レコードサイズフィールドは、FCB 内のオフセット 0EH にある値にセットされます。

AL に返されるコードは、以下の処理が行われたことを示します。

コード	意味
00H	リード (読み込み) が正しく行われ、処理完了した。
01H	ファイルの終わり。このレコードにデータは入っていない。
02H	ディスク転送アドレス内に、1レコードを読み込むのに十分な領域がなく、読み込みは取り消された。
03H	ファイルの終わり。〈EOF〉までのデータが読み込まれ、レコードの残りの部分がゼロで埋められた。

```

マクロ定義   read-seq macro fcb
                mov     dx, offset fcb
                mov     ah, 14H
                int      21H
                endm

```

例

つぎのプログラムは、ドライブ B: の TEXTFILE. ASC という名前のファイルをスクリーンに出力するものです。このファンクションは MS-DOS の TYPE コマンドに類似しています。読み込んだレコードの途中に EOF (エンドオブファイル: ASCII コード 1AH, <CTRL-Z>) がある場合、そこまでの文字がスクリーンに出力されます。

```

fcb          db      2, "TEXTFILEASC"
              db      25 dup (?)
buffer       db      128 dup (?), "$"
;
func_14 H:   set_dta   buffer          ; ディスク転送アドレスのセット (1AH)
              open     fcb              ; TEXTFILE.ASC ファイルをオープン (0AH)
read_line:   read_seq  fcb              ; シーケンシャルリード
              cmp      al, 02H          ; 読み込みが取り消されたか?
              je       all_done         ; はいの時, all_done へ
              cmp      al, 00H          ; ファイルエンドか?
              jg       check_more       ; はいの時, check_more へ
              display   buffer          ; buffer をスクリーンに出力 (09H)
              jmp      read_line        ; 次のレコードを得る
check_more:  cmp      al, 03H          ; レコードの残りが読み込まれているか?
              jne      all_done         ; いいえの時, all_done へ
              xor      si, si           ; インデックスを 0 にセット
find_eof:    cmp      buffer [si], 26  ; EOF キャラクタか?
              je       all_done         ; はいの時, all_done へ
              display_char buffer [si] ; バッファ中の文字をスクリーンに出力 (02H)
              inc      si               ; インデックスをインクリメント
              jmp      find_eof         ; 次の文字をチェック
all_done:    close     fcb              ; ファイルをクローズ (10H)

```


INT 21H

Sequential Write

ファンクション 15H

機能 シーケンシャルライト

コール AH = 15H
 DS : DX オープンされている FCB

リターン AL = 00H 正常な書き込み。
 01H ディスクに空き領域がない。
 02H ディスク転送アドレス (DTA)
 で示されるバッファが小さすぎる。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされている FCB のオフセット (DS にはセグメントアドレス) が入っていなければなりません。カレント (現在) のブロック (オフセット 0 CH) およびカレントのレコード (オフセット 20 H) フィールドによって示されているレコードにディスク転送アドレスから書き込まれ、つぎにカレントブロック、カレントレコードフィールドがつぎのレコードを示すようにセットされます。レコードサイズは、FCB 内のオフセット 0EH にある値にセットされます。レコードサイズが 1 セクタよりも小さい場合は、ディスク転送アドレスにあるデータがバッファに移され、このバッファに入れられたデータが 1 セクタに達した場合、ファイルがクローズされた場合またはリセットディスクシステムコール (ファンクション 0DH) が行われた場合、このバッファがディスクに書き込まれます。AL に返されるコードは、以下の処理が行われたことを示します。

コード	意味
00H	転送が正しく行われ、処理完了した。
01H	ディスクに空き領域がなく、書き込みは中止された。
02H	ディスク転送アドレスに、1 レコードを書き込むための十分な領域がないので、書き込みは中止された。

マクロ定義

```

write-seq macro fcb
            mov     dx, offset fcb
            mov     ah, 15H
            int     21H
        endm

```

例

つぎのプログラムは、ドライブ B: に DIR. TMP という名前のファイルを作成するものです。このファイルには、ディスクの番号(01H=A:, 02H=B:...)とファイル名が入っています。

```

rectrd_size equ 14 ; FCB 中のレコードサイズフィールドの
                   ; オフセット
;
fcb1 db 2, "DIR TMP"
      db 25 dup (?)
fcb2 db 2, "???????????"
      db 25 dup (?)
buffer db 128 dup (?)
;
func_15H: set_dta buffer ; ディスク転送アドレスのセット(1AH)
          search_first fcb2 ; 最初のエントリを検索
          cmp al, 0FFH ; ディレクトリエントリは存在するか?
          je all_done ; いいえの時, all_done へ
          create fcb1 ; DIR.TMP ファイルの作成(16H)
          mov fcb1 [record_size], 12
                                     ; レコードサイズ12 をセット
write_it: write_seg fcb1 ; シーケンシャルライト
          cmp al, 0
          jne all_done ; 正常終了なら all_done へ
          search_next fcb2 ; 次のエントリを検索
          cmp al, FFH ; エントリが存在しなければ all_done へ
          je all_done ; いいえの時, all_done へ
          jmp write_it ; はいの時, レコードをライト
all_done: close fcb1 ; ファイルをクローズ(10H)

```

INT 21H

Create File

ファンクション 16H

機能 ファイルの作成

コール AH = 16H
 DS : DX オープンされていない FCB

リターン AL = 00H 空のディレクトリが存在する。
 FFH 空のディレクトリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされていない FCB のオフセット (DS にはセグメントアドレス) が入っていなければなりません。空のエントリまたは指定されたファイル名の既存のエントリを見つけるために、ディレクトリが検索されます。

空のディレクトリエントリが存在する場合、このエントリはファイルサイズゼロに初期値設定され、オープンファイルファンクション (0FH) が行われ、AL に 00H が返されます。アトリビュートバイト (オフセット FCB-1) を 2 にセットした拡張 FCB を使用すると、隠されたファイルを作成することができます。

指定されたファイル名のエントリが存在する場合、このファイル名に対しオープンファイルシステムコール (ファンクション 0FH) が行われます。(すなわち、既存のファイルは消去され、新規の空のファイルが作成されることになります。)

空のディレクトリエントリも指定されたファイル名のエントリも存在しない場合、AL に FFH (255) が返されます。

マクロ定義

```
create macro fcb
    mov     dx, offset fcb
    mov     ah, 16H
    int     21H
endm
```


例

つぎのプログラムは、ドライブ B: に DIR.TMP という名前のファイルを作成するものです。このファイルには、ディスクの番号(01H=A:, 02H=B: ...) とこのディスク上のファイル名が入ります。

```

record-size equ      14                ;FCBのレコードサイズフィールドの
                                         ;オフセット
;
fcb1          db      2, "DIR    TMP"
              db      25 dup (?)
fcb2          db      2, "???????????"
              db      25 dup (?)
buffer        db      128 dup (?)
;
func_16H:     set_dta    buffer          ;ディスク転送アドレスのセット(1AH)
              search_first fcb2          ;最初のエントリを検索(11H)
              cmp        al, 0FFH        ;ディレクトリエントリが存在するか?
              je         all_done         ;いいえの時, all_doneへ
              create     fcb1            ;DIR.TMP ファイルの作成
              mov        fcb1 [record-size], 12
                                         ;レコードサイズ12をセット
write_it:     write_seq  fcb1            ;シーケンシャルライト(15H)
              search_next fcb2          ;次のエントリを検索(12H)
              cmp        al, 0FFH        ;ディレクトリエントリが存在するか?
              je         all_done         ;いいえの時, all_doneへ
              jmp        write_it        ;はいの時, レコードをライト
all_done:     close     fcb1            ;ファイルをクローズ(10H)

```

INT 21H

Rename File

ファンクション 17H

機能 ファイル名の変更

コール AH = 17H
DS : DX 修正された FCBリターン AL = 00H ディレクトリエントリが存在する。
FFH 目的のディレクトリエントリが存在しないか、またはファイル名がすでに存在している。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、FCB のオフセット (DS にはセグメントアドレス) が入っていないければなりません。この FCB にはドライブ番号とファイル名につづいて、オフセット 11H から新しいファイル名が入っていないければなりません。修正したいファイル名 (ワイルドカード文字を使うことができます) と一致するエントリを探すために、このディスクディレクトリが検索されます。

一致するディレクトリエントリが存在し、かつ 2 番目のファイル名が存在しない場合、ディレクトリエントリのファイル名は、修正用 FCB の新しいファイル名に変更されます (新旧のファイル名が、同じであってはなりません)。新しい 2 番目のファイル名にワイルドカード文字 "?" が使用されている場合、古いファイル名の対応する文字は変更されません。処理が完了すると AL に 00H が返されます。

このファンクションは隠しファイルまたはシステムファイルは使用できません。一致するディレクトリエントリが存在しないか、または 2 番目のファイル名のエントリがすでに存在する場合は、AL に FFH (255) が返されます。

マクロ定義

```

rename macro fcb
    mov     dx, offset fcb
    mov     ah, 17H
    int     21H
endm

```

例

つぎのプログラムは、変更したいファイル名と新しいファイル名の入力するプロンプトを出力し、ファイル名の変更を行うものです。

```

fcb          db      37 dup (?)
prompt1      db      "Filename: $"
prompt2      db      "New name: $"
reply        db      17 dup (?)
crlf         db      13, 10, "$"
;
func_17 H:   display  prompt1                ; prompt1 をスクリーンに出力(09H)
              get_string 15, reply            ; バッファードキーボード入力(0AH)
              display  crlf                  ; crlf をスクリーンに出力(09H)
              parse    reply [2], fcb        ; ファイル名の解析(29H)
              display  prompt2              ; prompt2 をスクリーンに出力(09H)
              get_string 15, reply            ; バッファードキーボード入力(0AH)
              display  crlf                  ; crlf をスクリーンに出力(09H)
              parse    reply [2], fcb [16]   ; ファイル名の解析(29H)
              rename   fcb                  ; ファイル名の変更

```


INT 21H

Get Current Disk

ファンクション 19H

- 機 能** カレントディスクを得る
- コール** AH = 19H
- リターン** AL 現在選択されているドライブ
 (00H = A, 01H = B, ...)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSh		FLAGSL
CS		
DS		
SS		
ES		

- 解 説** AL に現在選択されているドライブ(00H=A:, 01H=B:, ...)が返されます。

マクロ定義

```
current_disk macro
    mov ah, 19H
    int 21H
endm
```

- 例** つぎのプログラムは、2ドライブシステムで現在選択されている(カレント)ドライブをスクリーンに表示するものです。

```
message db "Current disk is $" ; $の説明は
crlf db 13, 10, "$" ; ファンクション 09H を参照
;
func_19H display message ; message をスクリーンに出力
current_disk ; カレントドライブを得る
cmp al, 00H ; カレントドライブは A か?
jne disk_b ; いいえの時, disk_b へ
display_char "A" ; "A" をスクリーンに出力
```

```

                jmp    all_done
disk_b:         display_char "B"           ; "B"をスクリーンに出力(02H)
all_done:       display    crlf           ; crlf をスクリーンに出力(09H)
    
```

INT 21H

Set Disk Transfer Address

ファンクション 1AH

機 能	ディスク転送アドレスのセット
コール	AH = 1AH DS : DX ディスク転送アドレス
リターン	なし

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解 説	DX には、ディスク転送アドレスのオフセット (DS にはセグメントアドレス) が入っていなければなりません。セグメントの終了から先頭へ向うディスク転送や、他のセグメントにオーバーフローするようなディスク転送は許されていません。
-----	--

注意：

ディスク転送アドレスをセットしない場合、MS-DOS ではプログラムセグメントプレフィクス内のオフセット 80H をデフォルト値として使用します。

カレントディスク転送アドレスはファンクション 2FH で得ることができます。

マクロ定義	<pre> set_dta macro buffer mov dx, offset buffer mov ah, 1AH int 21H endm </pre>
-------	--

例	つぎのプログラムは、プロンプトを出力し、入力したアルファベットを数字 (A=01H, B=02H, ...) に変換し、つぎにドライブ B : ALPHABET.DAT と
---	--

いう名前のファイルから対応するレコードを読み出し、それをスクリーンに表示するものです。このファイルには各レコードのサイズは28バイトの長さの26個のレコードが入っています。

```

record_size    equ    14                ; FCB のレコードの
                                           ; サイズフィールドのオフセット
relative_record equ    33                ; FCB のレコードの
                                           ; 相対レコードフィールドのオフセット
;
fcb             db      2, "ALPHABETDAT"
               db      25 dup (?)
buffer          db      34 dup (?), "$"
prompt          db      "Enter letter: $"
crlf            db      13, 10, "$"
;
func_1AH:       set_dta  buffer          ; ディスク転送アドレスのセット
               open    fcb              ; ALPHABET.DAT ファイルの
                                           ; オープン(0FH)
               mov     fcb [record_size], 28 ; レコードサイズ28 をセット
get_char:       display prompt          ; prompt をスクリーンに出力(09H)
               read_kbd_and_echo        ; キーボード入力(01H)
               cmp     al, 0DH           ; キャリッジリターンか?
               je      all_done         ; はいの時, all_done へ
               sub     al, 41H           ; いいえの時,
                                           ; ASCII コードをレコード番号に変換
               mov     fcb [relative_record], al
                                           ; 対応するレコードをセット
               display crlf             ; crlf をスクリーンに出力
               read_ran fcb             ; ALPHABET.DAT ファイルをランダム
                                           ; リード(21H)
               display buffer          ; buffer をスクリーンに出力(09H)
               display crlf            ; crlf をスクリーンに出力(09H)
               jmp     get_char         ; 次の文字を得る
all_done:       close    fcb            ; ファイルをクローズ(15H)

```

INT 21H

Get Default Drive Data

ファンクション 1BH

機能 デフォルトドライブのデータを得る

コール AH = 1BH

リターン AL 1 クラスタ当りのセクタ数
 CX 1 セクタ当りのバイト数
 DX 1 ドライブ当りのクラスタ数
 DS : BX FAT-ID のアドレス

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 データはレジスタに返されます。つぎにその内容を示します。

AL 1 クラスタ当りのセクタ数 (アロケーションユニット)
 CX 1 セクタ当りのバイト数
 DX デフォルトドライブのクラスタ数

BX は、ドライブのタイプを表すファイルアロケーションテーブル (FAT) の最初の 1 バイトのオフセットアドレス (DS は、セグメントアドレス) を返します。つぎにその内容を示します。

値	ドライブのタイプ
FFH	320KB ディスク, 1トラック 8 セクタ
FEH	256KB ディスク, 1トラック 26 セクタ
	1MB ディスク, 1トラック 8 セクタ
	160KB ディスク, 1トラック 8 セクタ
FDH	320KB ディスク, 1トラック 9 セクタ
FCH	160KB ディスク, 1トラック 9 セクタ
FBH	640KB ディスク, 1トラック 8 セクタ
F9H	640KB ディスク, 1トラック 9 セクタ
FEH	ハードディスク

似た機能を持つファンクションとして2つのファンクションが挙げられます。1つはファンクション 36H (ディスクのフリースペースを得る) で、相違点は、DX が返す値が、FAT-ID のアドレスではなく、使用可能なクラスタ数だけです。もう1つは、ファンクション 1CH (ドライブのデータを得る) で、デフォルト以外のディスクのデータを得ることができます。

ファイルアロケーションテーブルを含む MS-DOS のディスクデータの詳細に関しては第3章を参照してください。

マクロ定義

```
def_drive_data    macro
                  push        ds
                  mov         ah, 1BH
                  int         21H
                  mov         al, byte ptr [bx]
                  pop         ds
                  endm
```

例

つぎのプログラムはデフォルトのドライブが 640KBFD か別のディスクドライブかを判別します。

```
stdout            equ        1
;
msg               db          "Default drive is"
other             db          "another."
fd640             db          "fd640."
crlf              db          0DH, 0AH
;
func_1BH:         write_handle stdout, msg, 17      ; メッセージ表示
                  jc          write_error           ; エラー処理へ
                  def_drive_data                   ; デフォルトドライブのデータを得る
                  cmp         byte ptr[bx], 0FBH   ; FAT ID のリターン値= 0FBH か?
                  jne         diskette              ; いいえの時, diskette へ
                  write_handle stdout, fd640, 6    ; fd640 を表示(40H)
                  jc          write_error           ; エラーの時 write_error へ
                  jmp short   all_done              ; クリア & all_done へ
diskette:         write_handle stdout, other, 8     ; other を表示(40H)
all_done:         write_handle stdout, crlf, 2      ; crlf を表示(40H)
                  jc          write_error           ; エラー処理へ
```


INT 21H

Get Drive Data

ファンクション 1CH

機 能	ドライブのデータを得る
コール	AH = 1CH DL ドライブ番号 (00H = カレント, 01H = A, ……)
リターン	AL = FFH ドライブ番号の指定が無効 FFH 以外 1 クラスタ当りのセクタ数 CX 1 セクタ当りのバイト数 DX 1 ドライブ当りのクラスタ数 DS : BX FAT-ID のアドレス

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSh		FLAGSl
CS		
DS		
SS		
ES		

解 説 DL で指定されたドライブ (00H=カレント, 01H=A, ……) のデータを返します。データはつぎのレジスタによって返されます。

AL 1 クラスタ (アロケーションユニット) 当りのセクタ数
CX 1 セクタ当りのバイト数
DX ドライブのクラスタ数

BX は、ドライブのタイプを表すファイルアロケーションテーブル (FAT) の最初の 1 バイトのオフセットアドレス (DS は、セグメントアドレス) を返します。つぎにその内容を示します。

値	ドライブのタイプ
FFH	320KB ディスク, 1トラック 8 セクタ
FEH	256KB ディスク, 1トラック 26 セクタ
	1MB ディスク, 1トラック 8 セクタ
	160KB ディスク, 1トラック 8 セクタ
FDH	320KB ディスク, 1トラック 9 セクタ
FCH	160KB ディスク, 1トラック 9 セクタ
FBH	640KB ディスク, 1トラック 8 セクタ
F9H	640KB ディスク, 1トラック 9 セクタ
FEH	ハードディスク

DL で指定されたドライブ番号が無効の場合は、AL に FFH を返します。

似た機能を持つファンクションとして2つのファンクションが挙げられます。

1つは、ファンクション 36H (ディスクのフリースペースを得る) で、相違点は、BX が返す値が、FAT-ID のアドレスではなく、使用可能なクラスタ数であることです。もう1つは、ファンクション 1BH (デフォルトドライブのデータを得る) で、デフォルトのディスクのみのデータを得ることです。

ファイルアロケーションテーブルを含む MS-DOS のディスクデータの詳細に関しては第3章を参照してください。

マクロ定義

```
drive_data      macro      drive
                  push      ds
                  mov        dl, drive
                  mov        ah, 1CH
                  int         21H
                  mov        al, byte ptr [bx]
                  pop        ds
                  endm
```

例

つぎのプログラムは、ドライブ B: が 640KBFD か別のディスクドライブかを判別します。

```
stdout          equ        1
;
msg              db         "Drive B is"
other            db         "another."
fd640            db         "fd640 "
crlf             db         0DH, 0AH
;
begin            write handle stdout, msg, 11      ; msg を表示
                  jc         write_error           ; エラー処理へ
                  drive_data 2                    ; ドライブのデータを得る
                  cmp        byte ptr[bx], 0FBH    ; FAT ID のリターン値= 0FBH か?
                  jne        diskette              ; いいえの時, diskette へ
                  write handle stdout, fd640, 6    ; fd640 を表示(40H)
                  jc         write_error           ; エラー処理へ
                  jmp        all_done              ; クリア & all_done へ
```

```
diskette:    write_handle  stdout, other, 8    ; other を表示(40H)
all_ done:   write_handle  stdout, crlf, 2    ; crlf を表示(40H)
jc           write_error    ; エラー処理へ
```


INT 21H

Random Read

ファンクション 21H

機能 ランダムリード

コール AH = 21H
DS : DX オープンされた FCB

リターン AL = 00H 正常な読み込み
01H EOF
02H ディスク転送アドレス (DTA)
で示されるバッファが小さすぎる。
03H EOF, レコードの一部

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされた FCB のオフセット (DS にはセグメントアドレス) が入っていなければなりません。カレント (現在の) ブロック (オフセット 0CH) とカレントレコード (オフセット 20H) フィールドが、相対レコードフィールド (オフセット 21H) と一致するようにセットされ、つぎにこれらのフィールドによって指定されたレコードが、ディスク転送アドレスにロードされます。

AL に返されるコードは、つぎの処理が行われたことを示します。

コード	意味
00H	リード (読み込み) が正しく行われ、処理が完了した。
01H	ファイルの終わり。このレコードにデータは存在しない。
02H	ディスク転送アドレスに、1レコードを読み込むだけの十分な領域がなく、読み込みは中止された。
03H	ファイルの終わり。〈EOF〉までのデータが読み込まれ、レコードの残りの部分がゼロで埋められた。

マクロ定義 read_ran

```

macro fcb
mov dx, offset fcb
mov ah, 21H
int 21H
endm

```

例

つぎのプログラムは、文字入力を促すプロンプトを表示し、入力したアルファベットを数字(A=01H, B=02H, C=03H, ...)に変換し、次にドライブ B: の ALPHABET.DAT という名前のファイルから対応するレコードを読み出し、それをスクリーンに出力するものです。このファイルには、各レコードは 28 バイトの長さの 26 個のレコードが入っています。

```
record_size    equ    14                ; FCB の
                                           ; サイズフィールドのオフセット
relative_record equ    33              ; FCB の
                                           ; 相対レコードフィールドのオフセット
;
fcb            db      2, "ALPHABETDAT"
               db      25 dup (?)
buffer         db      34 dup (?), "$"
prompt        db      "Enter letter: $"
crlf           db      13, 10, "$"
;
func_21H:      set_dta buffer           ; ディスク転送アドレスのセット(1AH)
               open    fcb             ; ALPHABET.DAT ファイルのオープン
                                           ; (0FH)

               mov     fcb [record_size], 28 ; レコードサイズ 28 をセット
get_char:      display prompt           ; prompt をスクリーンに出力(09H)
               read_kbd_and_echo       ; キーボード入力(01H)
               cmp     al, 0DH          ; キャリッジリターンか?
               je      all_done         ; はいの時, all_done へ
               sub     al, 41H          ; いいえの時, ASCII コードを
                                           ; レコード番号に変換

               mov     fcb [relative_record], al ; 対応するレコードをセット
               display crlf            ; crlf をスクリーンに出力(09H)
               read_ran fcb           ; ALPHABET.DAT ファイルを
                                           ; ランダムリード

               display buffer          ; buffer をスクリーンに出力(09H)
               display crlf            ; crlf をスクリーンに出力(09H)
               jmp     get_char         ; 次の文字を得る
all_done:      close fcb               ; ファイルをクローズ(10H)
```

INT 21H

Random Write

ファンクション 22H

機能 ランダムライト

コール AH = 22H

DS : DX オープンされた FCB

リターン AL = 00H 正常な書き込み
 01H ディスクに空き領域がない。
 02H ディスク転送アドレス (DTA)
 で示されるバッファが小さすぎる。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされた FCB 内のオフセット (DS にはセグメントアドレス) が入っていなければなりません。カレント (現在) のブロック (オフセット 0CH) とカレントレコード (オフセット 20H) フィールドが相対レコードフィールド (オフセット 21H) と一致するようにセットされ、つぎにこれらのフィールドによって指定されたレコードへ、ディスク転送アドレスから書き込まれます。レコードサイズが 1 セクタより小さい場合は、これらのレコードがバッファに入れられ、1 セクタ分に達したとき、ファイルがクローズされた、ファンクション 0DH が実行された、のいずれかの場合、書き込まれます。

AL に返されるコードは、以下の処理が行われたことを示します。

コード	意味
00H	書き込みが正しく行われ、処理は完了した。
01H	ディスクに空き領域がない。
02H	ディスク転送アドレスに、1 レコードを書き込みのに十分な領域がなく、書き込みは中止された。

マクロ定義

```

write_ran macro fcb
    mov     dx, offset fcb
    mov     ah, 22H
    int     21H
endm

```

例

つぎのプログラムは、文字入力を促すプロンプトを表示し、入力したアルファベットを数字 (A=01H, B=02H, C=03H, ...) に変換し、つぎにドライブ B: の ALPHABET.DAT という名前のファイルから対応するレコードを読み込み、それをスクリーンに出力するものです。このファイルには各レコード 28 バイトの長さの 26 個のレコードが入っています。該当するレコードを出力した後、変更されたレコードを入力するためのプロンプトを出力します。ユーザーが新規のレコードを入力した場合、そのレコードはファイルに書き込まれ、単にリターンキーのみを押した場合レコードの置換は行われません。

```

record_size equ 14 ; FCB のサイズフィールドのオフセット

relative_record equ 33 ; FCB の相対レコードフィールドのオフセット
;
fcb db 2, "ALPHABET.DAT"
    db 25 dup (?)
buffer db 26 dup (?), 13, 10, "$"
prompt1 db "Enter letter: $"
prompt2 db "New record (RETURN for no change) : $"
crlf db 13, 10, "$"
reply db 28 dup (32)
blanks db 26 dup (32)
;
func_22H: set_dta buffer ; ディスク転送アドレスのセット (1AH)
          open fcb ; ALPHABET.DAT ファイルのオープン (0FH)
          mov fcb [record_size], 28 ; レコードサイズ 28 をセット
get_char: display prompt1 ; prompt1 をスクリーンに出力 (09H)
          read_kbd_and_echo ; キーボード入力 (01H)

```

```

cmp    al, 0DH      ; キャリッジリターンか?
je     all_done     ; はいの時, all_done へ
sub    al, 41H      ; いいえの時, ASCII コードを
                    ; レコード番号に変換
mov    fcb [relative-record], al
                    ; 対応するレコードをセット
display crlf        ; crlf をスクリーンに出力(09H)
read_ran fcb        ; ランダムライト
display buffer      ; buffer をスクリーンに出力(09H)
display crlf        ; crlf をスクリーンに出力(09H)
display prompt2     ; prompt2 をスクリーンに出力(09H)
get_string 27, reply ; バッファードキーボード入力(0AH)
display crlf        ; crlf をスクリーンに出力(09H)
cmp    reply [1], 0 ; キャリッジリターン以外のキーが
                    ; 押されたか?
je     get_char     ; いいえの時,
                    ; 次の文字を得る
xor    bx, bx
mov    bl, reply [1] ; カウンタとして reply のバッファリングス
                    ; を使用
move_string blanks, buffer, 26 ; 章末参照
move_string reply [2], buffer, bx ; 章末参照
write_ran fcb        ; ランダムライト
jmp    get_char     ; 次の文字を得る
all_done: close fcb  ; ファイルをクローズ(10H)

```

INT 21H

Get File Size

ファンクション 23H

機能 ファイルの大きさを得る

コール AH = 23H
DS : DX オープンされていない FCB

リターン AL = 00H ディレクトリエントリが存在。
FFH ディレクトリエントリが存在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされていない FCB のオフセット (DS には、セグメントアドレス) が入っていなければなりません。このファンクションコールを行うためには、事前にレコードサイズフィールド (オフセット 0EH) を該当する値にセットしておきます。最初に一致するエントリを見つけるために、このディスクディレクトリが検索されます。

一致するディレクトリエントリが存在する場合、相対レコードフィールド (オフセット 21H) が、ディレクトリ内のファイルサイズ (オフセット 1CH) と FCB 内のレコードサイズフィールド (オフセット 0EH) から計算したファイルのレコード数にセットされ、AL に 00H が返されます。

一致するディレクトリが存在しない場合、AL に FFH (255) が返されます。

注意:

FCB のレコードサイズフィールド (オフセット 0EH) の値が、レコード内の実際の文字数と一致しない場合、このファンクションは正しいファイルサイズを返しません。デフォルトレコードサイズ (128) を使わない場合、このファンクションを使用する前にレコードサイズフィールドを正しい値にセットしておかなければなりません。

マクロ定義

```
file_size macro fcb
    mov     dx, offset fcb
    mov     ah, 23H
    int     21H
endm
```

例

つぎのプログラムは、ファイル名の入力を促すプロンプトを表示し、このファイルをオープンして FCB 内のレコードサイズフィールドを埋め、ファイルサイズシステムコールを行い、ファイルサイズとレコード数を 16 進でスクリーンに表示するものです。

```
fcb          db          37 dup (?)
prompt       db          "File name: $"
msg1         db          "Record length:  ", 13, 10, "$"
msg2         db          "Records:      ", 13, 10, "$"
crlf         db          13, 10, "$"
reply        db          17 dup(?)
sixteen      db          16
;
func_23H:    display     prompt          ; prompt をスクリーンに出力(09H)
             get_string  17, reply       ; バッファードキーボード入力(0AH)
             cmp         reply [1], 0    ; キャリッジリターンか?
             jne         get_length      ; いいえの時, get_length へ
             jmp         all_done         ; はいの時, all_done へ
get_length:  display     crlf            ; crlf をスクリーンに出力(09H)
             parse       reply [2], fcb  ; ファイル名の解析(29H)
             open        fcb             ; ファイルのオープン(0FH)
             file_size    fcb             ; ファイルの大きさを得る
             mov         si, 33          ; 相対レコードフィールドの
                                           ; オフセットをセット
             mov         di, 9           ; msg2 に答える
convert_it:  cmp         fcb [si], 0      ; 変換する数字か?
             je          show_it         ; いいえの時, show_it へ
             convert     fcb [si], sixteen, msg2 [di]
             inc         si              ; n-o-r インデックスをインクリメント
```

```
inc      di          ;メッセージインデックスを  
          ;インクリメント  
jmp      convert-it  ;数字をチェック  
show-it: convert fcb [14], sixteen, msg1 [15]  
          display msg1      ;msg1 をスクリーンに出力(09H)  
          display msg2      ;msg2 をスクリーンに出力(09H)  
          jmp      func_23H  ;別のファイル名を得る  
all-done: close fcb        ;ファイルをクローズ(10H)
```

INT 21H

Set Relative Record

ファンクション 24H

機能	相対レコードのセット
コール	AH = 24H DS : DX オープンされた FCB
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされた FCB のオフセット (DS には、セグメントアドレス) が入っていなければなりません。相対レコードフィールド (オフセット 21H) は、カレント (現在の) ブロック (オフセット 0CH) とカレントレコードフィールド (オフセット 20H) と同じファイルアドレスにセットされます。

ファンクション 21H, 22H, 27H, 28H を使用する前にこのファンクションを使ってファイルポインタをセットしなければなりません。

マクロ定義

```

set_relative_record macro fcb
    mov     dx, offset fcb
    mov     ah, 24H
    int     21H
endm

```


例

つぎのプログラムは、ランダムブロックリード（読み出し）とランダムブロックライト（書き込み）のシステムコールを使用して、ファイルのコピーを行うものです。レコードサイズをファイルの大きさと等しくなるようにセットし、レコードカウントを1にセットし、32K バイトのバッファを使用することによって、コピーの速度が速くなります。ファイルポインタは、カレントレコードフィールド（オフセット 20H）を1にセットし、相対レコードのセット機能を使用して相対レコードフィールド（オフセット 21H）をカレントブロック（オフセット 0CH）とカレントレコードフィールド（オフセット 20H）を組み合わせたものと同じレコードにポイントさせることによって、位置決めされます。

```

current-record equ 32          ;FCB のレコードサイズフィールドの
                                ;オフセット
file-size      equ 16          ;FCB のファイルサイズフィールドの
                                ;オフセット
;
fcb            db 37 dup (?)
filename       db 17 dup (?)
prompt1       db "File to copy: $" ;$の説明はファンクション 09H を参照
prompt2       db "Name of copy: $"
crlf          db 13, 10, "$"
file-length    dw ?
buffer        db 32767 dup(?)
;
func_24 H:    set_dta buffer      ;ディスク転送アドレスのセット
                                ;(1AH)
              display prompt 1    ;prompt1 をスクリーンに出力
                                ;(09H)
              get_string 15, filename ;ファイル名の入力(0AH)
              display crlf        ;crlf をスクリーンに出力(09H)
              parse filename [2], fcb ;ファイル名の解析(29H)
              open fcb            ;ファイルのオープン(0FH)
              mov fcb [current-record], 0 ;カレントレコードフィールド
                                ;をセット

```

```
set-relative-record fcb          ; 相対レコードをセット
mov ax, word ptr fcb [file_size]; ファイルサイズを得る
mov file_length, ax              ; ランダムブロックライトする
                                  ; ためにそれをセーブ
ran-block-read fcb, 1, ax        ; ランダムブロックリード(27H)
display prompt2                  ; prompt2 をスクリーンに出力
                                  ; (09H)
get-string 15, filename          ; ファイル名の入力(0AH)
display crlf                     ; crlf をスクリーンに出力(09H)
parse filename [2], fcb          ; ファイル名の解析(29H)
create fcb                       ; ファイルの作成(16H)
mov fcb [current-record], 0      ; カレントレコードフィールド
                                  ; をセット
set-relative-record fcb          ; 相対レコードをセット
mov ax, file_length              ; オリジナルファイルの長さを得る
ran-block-write fcb, 1, ax       ; ランダムブロックライト(28H)
close fcb                        ; ファイルのクローズ(10H)
```

INT 21H

Set Interrupt Vector

ファンクション 25H

機能	割り込みベクタのセット
コール	AH = 25H AL 割り込みタイプ番号 DS : DX 割り込み処理ルーチン
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説	<p>このファンクションは、特定の割り込みベクタをセットするために使用します。MS-DOS は、これによってプロセスごとに割り込みを管理することができます。</p> <p>指定した割り込みのベクタテーブルに、DS : DX で示される割り込み処理ルーチンのアドレスをセットします。DX には、割り込み処理ルーチンのオフセット (DS には、セグメントアドレス)が入っていないなければなりません。AL には、このルーチンによって処理される割り込みタイプの番号が入っていないなければなりません。</p>
-----------	---

マクロ定義	<pre> set_vector macro interrupt, seg_addr, off_addr push ds mov ax, seg_addr mov ds, ax mov dx, off_addr mov al, interrupt mov ah, 25H int 21H pop ds endm </pre>
--------------	---


```
例  lds    dx, intvector
     mov    ah, 25H
     mov    al, intnumber
     int     21H
     ;エラーがなければリターン
```

INT 21H

Create New PSP

ファンクション 26H

機 能 新しい PSP (プログラムセグメントプレフィクス) を作成する。

コール AH = 26H
DX 新しい PSP のセグメントアドレス

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 DX には必ず、新しい PSP のセグメントアドレスが入っていなければなりません。

このファンクションコールは、V2.0 以前の MS-DOS と互換性を保つために用意されています。新規のプログラムでは (V2.0 以前と互換性を保つ必要がないならば)、ファンクション 4BH, コード 00H を使って子プロセスを起動してください。

マクロ定義

```

create-psp      macro      seg-addr
                 mov        dx, seg-addr
                 mov        ah, 26H
                 endm
  
```

例 このファンクションは、ファンクション 4BH, コード 00H (プログラムのロードと実行), ファンクション 4BH, コード 03H (オーバーレイのロード) によって置き換えられますので、プログラムは省略いたします。

INT 21H

Random Block Read

ファンクション 27H

機能 ランダムブロックリード

コール AH = 27H
 DS : DX オープンされた FCB
 CX 読み出すべきレコード数

リターン AL = 00H 読み出しの正常終了
 01H EOF, 空レコード
 02H ディスク転送アドレス (DTA)
 で示されるバッファが小さすぎる。
 03H EOF, レコードの一部
 CX 読み取られたレコード数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 DX には、オープンされた FCB のオフセット (DS には、セグメントアドレス) が入っていなければなりません。CX には、読み出すべきレコード数をセットします。CX に 0 が入っている場合、レコードは読み出されずに (動作が行われないうちに)、このファンクションを終了します。指定されたレコード数 (レコードサイズフィールド (オフセット 0EH) から計算される) の読み出しが、相対レコードフィールド (オフセット 21H) で指定されたレコードから開始されます。読み出されたレコードは、ディスク転送アドレスに入ります。

このファンクション実行前にファンクション 24H によって相対レコードをセットしなければなりません。

AL に返されるレコードは、以下の処理が行われたことを示します。

コード	意味
00H	読み出しが正しく行われ、処理は完了した。
01H	ファイルの終わり、このレコードにデータは存在しない。
02H	ディスク転送アドレスに、1レコードを読み込むために十分な領域がなく、読み込みは中止された。
03H	ファイルの終わり、〈EOF〉までのデータが読み込まれ、レコードの残りの部分が 0 で埋められた。

CX に、読み取られたレコード数が返されます。カレントブロック (オフセット 0CH)、カレントレコード (オフセット 20H) および相対レコード (オフセット 21H) フィールドは、つぎのレコードのアドレスにセットされます。

マクロ定義

```
ran-block-read macro fcb, count, rec-size
    mov     dx, offset fcb
    mov     cx, count
    mov     word ptr fcb [14], rec-size
    mov     ah, 27H
    int     21H
endm
```

例

つぎのプログラムは、ランダムブロックリードファンクションを使用しファイルコピーを行うものです。レコードカウントを1に、またレコードサイズをファイルの大きさと等しくなるように指定し、32K バイトのバッファを使用することによって、コピーの速度が速くなります。このファイルは、1つのレコードとして読み込まれます (ファンクション 28H を使用したプログラム例と比較してください。ファンクション 28H では、レコードサイズを1に、レコードカウントをファイルの大きさと等しくなるように指定しています)。

```
current_record equ      32    ; カレントレコードフィールドのオフセット
file_size      equ      16    ; ファイルサイズフィールドのオフセット
;
fcb            db        37 dup (?)
filename       db        17 dup (?)
prompt1        db        "File to copy: $"    ; $の説明はファンクション
prompt2        db        "Name of copy: $"    ; 09H を参照
crlf           db        13, 10, "$"
file_length    dw        ?
buffer         db        32767 dup (?)
;
func-27H:      set-dta      buffer    ; ディスク転送アドレスのセット (1AH)
               display     prompt1    ; prompt1 をスクリーンに出力 (09H)
               get         string 15, filename ; ファイル名の入力 (0AH)
```

display	crlf	; crlf をスクリーンに出力(09H)
parse	filename [2], fcb	; ファイル名の解析(29H)
open	fcb	; ファイルのオープン(0FH)
mov	fcb[current-record], 0	; カレントレコードフィールド ; に 0 をセット
set-relative-record	fcb	; 相対レコードをセット(24H)
mov	ax, word ptr fcb [file-size]	; ファイルサイズを得る
mov	file-length, ax	; ランダムブロックライトのた ; めにそれをセーブする
ran-block-read	fcb, 1, ax	; ランダムブロックリード
display	prompt2	; prompt2 をスクリーンに出力 ; (09H)
get-string	15, filename	; ファイル名の入力(0AH)
display	crlf	; crlf をスクリーンに出力(09H)
parse	filename [2], fcb	; ファイル名の解析(29H)
create	fcb	; ファイルの作成(16H)
mov	fcb[current-record], 0	; カレントレコードフィールド ; に 0 をセット
set-relative-record	fcb	; 相対レコードをセット(24H)
mov	ax, file-length	; オリジナルファイルのサイズ ; を得る
ran-block-write	fcb, 1, ax	; ランダムブロックライト(28H)
close	fcb	; ファイルをクローズ(10H)

INT 21H

Random Block Write

ファンクション 28H

機 能 ランダムブロックライト

コール AH = 28H
 DS : DX オープンされた FCB
 CX 書き込むべきレコード数 (0 = ファイルサイズフィールドをセットします。)

リターン AL = 00H 書き込みの正常終了
 01H ディスクの空き領域がない。
 02H ディスク転送アドレス (DTA) で示されるバッファが小さすぎる。
 CX 書き込まれたレコード数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 DX には、オープンされた FCB のオフセット (DS には、セグメントアドレス) が、CX には、書き込むべきレコード数または 0 が入っていなければなりません。指定されたレコード数 (オフセット 0EH のレコードサイズフィールドから計算した) が、ディスク転送アドレスから書き込まれます。ファイルへのレコードの書き込みは、FCB の相対レコードフィールド (オフセット 21H) が指定されたレコードから開始されます。CX が 0 の場合、レコードは書き込まれませんが、ディレクトリエントリのファイルサイズフィールド (オフセット 1CH) が、FCB の相対レコードフィールド (オフセット 21H) で指定されたレコード数にセットされます。アロケーションユニットは、必要に応じ割り当てられるか、または解放されます。

このファンクション実行前にファンクション 24H によって相対レコードをセットしなければなりません。

AL に返されるコードは、以下の処理が行われたことを示します。

コード	意味
00H	書き込みが正しく行われ、処理が完了した。
01H	ディスクに、空き領域がない。
02H	ディスク転送アドレスに、1 レコードを読み込むのに十分な領域がなく、読み込みは中止された。

CX には、書き込まれたレコード数が返されます。カレントブロック (オフセット 0CH)、カレントレコード (オフセット 20H) および相対レコード (オフセット 21H) の各フィールドは、そのつぎのレコードアドレスにセットされます。

マクロ定義

```
ran_block_write macro fcb, count, rec_size
    mov     dx, offset fcb
    mov     cx, count
    mov     word ptr fcb [14], rec_size
    mov     ah, 28H
    int     21H
endm
```

例

つぎのプログラムは、ランダムブロックリードおよびランダムブロックライトのファンクションを使用してファイルをコピーするものです。レコードカウントをファイルの大きさと等しくなるように指定し、レコードサイズを1に指定し、また 32K バイトのバッファを使用することによって、コピーの速度を速くすることができます。ファイルの読み出しおよび書き込みは1回のディスクアクセスで行われるので、迅速にコピーされます(ファンクション 27H のプログラム例と比較してください。ファンクション 27H では、レコードのカウントが1に、またレコードサイズがファイルの大きさと等しくなるように指定されています)。

```
current_record equ 32    ; カレントレコードフィールドのオフセット
file_size      equ 16    ; ファイルサイズフィールドのオフセット
;
fcb            db        37 dup (?)
filename       db        17 dup (?)
prompt1       db        "File to copy: $"          ; $の説明はファンクション
prompt2       db        "Name of copy: $"          ; 09H を参照
crlf          db        13, 10, "$"
num_recs      dw        ?
buffer        db        32767 dup (?)
;
func_28H: set_dta    buffer                        ; ディスク転送アドレスのセット(1AH)
            display   prompt1                      ; prompt1 をスクリーンに出力(09H)
```

```

get_string 15, filename      ; ファイル名の入力(0AH)
display     crlf             ; crlf をスクリーンに出力(09H)
parse       filename [2], fcb ; ファイル名の解析(29H)
open        fcb              ; ファイルのオープン(0FH)
mov         fcb [current-record], 0 ; カレントレコードフィールドに 0 をセット
set_relative_record fcb      ; 相対レコードをセット(24H)
mov         ax, word ptr fcb [file-size]
                                   ; ファイルサイズを得る
mov         num-recs, ax      ; ランダムブロックライトのために
                                   ; それをセーブ
ran_block_read fcb, num-recs, 1 ; ランダムブロックリード(27H)
display     prompt2          ; prompt2 をスクリーンに出力(09H)
get_string 15, filename      ; ファイル名の入力(0AH)
display     crlf             ; crlf をスクリーンに出力(09H)
parse       filename [2], fcb ; ファイル名の解析(29H)
create      fcb              ; ファイルの作成(16H)
mov         fcb [current-record], 0 ; カレントレコードフィールドに 0 をセット
set_relative_record fcb      ; 相対レコードをセット(24H)
mov         ax, file-length   ; オリジナルのサイズを得る
ran_block_write fcb, num-recs, 1 ; ランダムブロックライト
close       fcb              ; ファイルをクローズ(10H)

```

INT 21H

Parse File Name

ファンクション 29H

機能 ファイル名の解析

コール AH = 29H

AL 解析の制御（解説を参照してください）。

DS : SI 解析すべきストリング

ES : DI オープンされていない FCB

リターン AL = 00H ワイルドカード文字が、使用されていない。

01H ワイルドカード文字が、使用されている。

FFH ドライブ文字が無効。

DS : SI 解析されたストリングのつぎにくる最初のバイト。

ES : DI オープンされていない FCB

解説 SI には、解析すべきストリング（コマンド行）のオフセット（DS には、セグメントアドレス）が、DI には、オープンされていない FCB のオフセット（ES には、セグメントアドレス）が入っていなければなりません。d: ファイル名, 拡張子という書式のファイル名を見つけるために、このストリングを解析し、このファイル名が存在する場合、対応するオープンされていない FCB が ES : DI に作成されます。

AL レジスタの 0～3 ビット目は、解析処理を制御するためのものです。4～7 ビット目は、無視されます。

ビット	値	意味
0	0	ファイル分離記号を検出した場合、すべての解析を停止。
	1	先行する分離記号を無視。
1	0	ストリングにドライブ番号が入っていない場合、FCB 内のドライブ番号は 0（カレントドライブ）にセットされる。
	1	ストリングにドライブ番号が入っていない場合、FCB 内のドライブ番号は変更されない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

- | | | |
|---|---|--|
| 2 | 1 | ストリングにファイル名が入っていない場合、FCB 内のファイル名は変更されない。 |
| | 0 | ストリングにファイル名が入っていない場合、FCB 内のファイル名は 8 つのスペースにセットされる。 |
| 3 | 1 | ストリングに拡張子が入っていない場合、FCB 内の拡張子は変更されない。 |
| | 0 | ストリングに拡張子が入っていない場合、FCB 内の拡張子は 3 つのスペースにセットされる。 |

ファイル名または拡張子にアスタリスク（*）が入っている場合、ファイル名または拡張子内の他のすべての文字は疑問符（?）にセットされます。

ファイル名分離記号をつぎに示します。

: . ; , = + / " [] ¥ < > | スペース タブ

ファイル名の終了記号には、すべてのファイル名の分離記号とすべての制御文字が含まれます。ファイル名の中にファイル名の終了記号を入れることはできません。終了記号を検出した場合、解析は停止します。

ストリングに有効なファイル名が入っている場合

- | | |
|---|---|
| 1 | ファイル名または拡張子にワイルドカード文字（*または?）が入っている場合は AL に 1 が、入っていない場合 0 が AL に返される。 |
| 2 | DS:SI は、解析されたストリングのつぎにくる最初の文字を示す。
ES:DI は、オープンされていない FCB の先頭のバイトを示す。 |

ドライブ名が無効な場合、AL に FFH (255) が返されます。ストリングに有効なファイル名が入っていない場合、ES:DI+1 は、スペース(ASCII コード 32)を示します。

マクロ定義

```

parse macro string, fcb
    mov     si, offset string
    mov     di, offset fcb
    push    es
    push    ds
    pop     es
    mov     al, 0FH ; 0, 1, 2, 3 のビットが ON である
    mov     ah, 29H
    int     21H
    pop     es
endm

```

例

つぎのプログラムは、プロンプトに回答して入力された名前のファイルが存在するかどうか検索するものです。

```

fcb          db      37 dup (?)
prompt       db      "Filename: $"
reply        db      17 dup (?)
yes          db      "FILE EXISTS", 13, 10, "$"
no           db      "FILE DOES NOT EXIST", 13, 10, "$"
;
func_29H:    display  prompt          ; prompt をスクリーンに出力(09H)
             get_string 15, reply      ; ファイル名の入力(0AH)
             parse      reply [2], fcb ; ファイル名の解析
             search_first fcb          ; 最初のエントリを検索(11H)
             cmp        al, 0FFH       ; ディレクトリエントリが存在するか?
             je         not_there      ; いいえの時, not_there へ
             display    yes           ; はいの時, yes をスクリーンに出力(09H)
             jmp        continue
not_there:   display    no
continue:    .
             .

```

INT 21H

Get Date

ファンクション 2AH

機能	日付を得る
コール	AH = 2AH
リターン	CX 年 (1980~2099) DH 月 (1~12) DL 日 (1~31) AL 曜日 (0 = 日, 1 = 月, ...6 = 土)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解説	CX および DX に、MS-DOS で管理している現在の日付が2進数で返されます。
----	--

CX 年 (1980~2099)
 DH 月 (1=1月, 2=2月, ...)
 DL 日 (1~31)
 AL 曜日 (0=日曜日, 1=月曜日, ...)

マクロ定義	<pre> get-date macro mov ah, 2AH int 21H endm </pre>
-------	--

例	つぎのプログラムは、日付を取得し、その翌日に更新されます。必要な場合、月または年を1つ増やし、新規の日付にセットするものです。
---	---

```

month db 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
;

```



```

func_2AH:  get-date          ; 日付を得る
            inc dl           ; 日をインクリメント
            xor bx, bx       ; BL はインデックスとして使用
            mov bl, dh       ; 月をインデックスレジスタにセット
            dec bx
            cmp dl, month [bx] ; 月の最後の日を越えているか?
            jle month_ok     ; いいえの時, 新規の日付をセット, month_ok へ
            mov dl, 1        ; はいの時, 日を1にセット
            inc dh           ; そして, 月をインクリメント
            cmp dh, 12       ; 年の最後の月を越えているか?
            jle month_ok     ; いいえの時, 新規の日付をセット, month_ok へ
            mov dh, 1        ; はいの時, 月を1にセット
            inc cx           ; 年をインクリメント
month_ok:   set_date cx, dh, dl ; 日付のセット (2BH)

```

INT 21H

Set Date

ファンクション 2BH

機能 日付をセットする

コール AH = 2BH

CX 年 (1980~2099)

DH 月 (1~12)

DL 日 (1~31)

リターン AL = 00H 有効な日付
 FFH 無効な日付

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 CX および DX レジスタには、2進で表した有効な日付が入っていなければなりません。

CX 年 (1980~2099)

DH 月 (1=1月, 2=2月, ...)

DL 日 (1~31)

日付が有効な場合、この日付がセットされ AL に 00H が返されます。無効な場合、このファンクションは中止され AL に FFH (255) が返されます。

マクロ定義

```

set_date macro year, month, day
    mov     cx, year
    mov     dh, month
    mov     dl, day
    mov     ah, 2BH
    int     21H
endm

```

例 つぎのプログラムは、日付を取得し、その翌日に更新します。必要な場合日付月または年を1つ増やし、新しい日付にセットします。

```
month    db    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
;
func_2BH: get_date          ; 日付を得る(2AH)
          inc    dl          ; 日をインクリメント
          xor    bx, bx      ; BL はインデックスとして使用
          mov    bl, dh      ; 月をインデックスレジスタにセット
          dec    bx
          cmp    dl, month [bx] ; 月の最後の日を越えているか?
          jle    month_ok    ; いいえの時, 新規の日付をセット, month_ok へ
          mov    dl, 1       ; はいの時, 日を1にセット
          inc    dh          ; そして, 月をインクリメント
          cmp    dh, 12      ; 年の最後の月を越えているか?
          jle    month_ok    ; いいえの時, 新規の日付をセット month_ok へ
          mov    dh, 1       ; はいの時, 月を1にセット
          inc    cx          ; 年をインクリメント
month_ok: set_date cx, dh, dl ; 日付のセット
```


INT 21H

Get Time

ファンクション 2CH

機能 時刻を得る

コール AH = 2CH

リターン CH 時 (0~23)
 CL 分 (0~59)
 DH 秒 (0~59)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 MS-DOS が管理する時刻を 2 進数表現で, CX および DX に返します。

CH 時 (0~23)
 CL 分 (0~59)
 DH 秒 (0~59)

マクロ定義 `get_time macro`
`mov ah, 2CH`
`int 21H`
`endm`

例 つぎのプログラムは, 任意のキーが入力されるまで時刻をスクリーンに継続的に出力するものです。

```
time      db      "00:00:00.00", 13, 10, "$"
ten       db      10
```

・
 ・

```
func_2CH: get_time          ;時刻を得る(このファンクションの実行)
```

```

convert ch, ten, time      ; 章末参照
convert cl, ten, time [3]  ; 章末参照
convert dh, ten, time [6]  ; 章末参照
display time               ; 時刻をスクリーンに出力(09H)
check_kbd_status           ; キーボードステータスの検査(0BH)
cmp    al, 0FFH            ; キー入力されたか?
je     all_done             ; はいの時, 処理終了
jmp    func_2CH            ; いいえの時, 時刻の表示を継続

all_done:

```

INT 21H

Set Time

ファンクション 2DH

機能 時刻をセットする

コール AH = 2DH
 CH 時 (0~23)
 CL 分 (0~59)
 DH 秒 (0~59)
 DL 0

リターン AL = 00H 有効な時刻
 FFH 無効な時刻

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 CX および DX レジスタには、2進で表した有効な時刻が入っていなければなりません。DL = 00 H でなければなりません。

CH 時 (0~23)
 CL 分 (0~59)
 DH 秒 (0~59)

指定された時刻が有効な場合、その時刻がセットされ、AL に 00H が返されます。無効な場合、このファンクションは中止され、AL に FFH (255) が返されます。

マクロ定義

```

set_time macro hour, minutes, seconds
    mov     ch, hour
    mov     cl, minutes
    mov     dh, seconds
    mov     dl, 0
    mov     ah, 2DH
    int     21H
endm

```


例

つぎのプログラムは、システムクロックを0にセットしたのち、時刻を継続的にスクリーンに出力するものです。任意のキーが入力されると、時刻の表示が停止し、再びキーが入力されると、クロックは0にリセットされ、時刻の表示が再開します。

```

time          db  "00:00:00.00", 13, 10, "$"
ten           db  10
;
func_2DH:     set_time  0, 0, 0           ;時刻をセット
read_clock:   get_time                    ;時刻を得る(2CH)
              convert  ch, ten, time      ;章末参照
              convert  cl, ten, time [3]  ;章末参照
              convert  dh, ten, time [6]  ;章末参照
              display  time                ;時刻をスクリーンに出力(09H)
              dir_console_io  0FFH        ;キー入力(06H)
              cmp      al, 00H            ;文字は入力されたか?
              jne      stop               ;はいの時、時刻の表示を停止, stopへ
              jmp      read_clock         ;いいえの時、時刻の表示を継続
stop:         read_kbd                    ;キーの再入力(08H)
              jmp      func_2DH           ;時刻の表示を再開

```

INT 21H

Set/Reset Verify Flag

ファンクション 2EH

機能 ベリファイフラグのセット/リセット

コール AH = 2EH
 AL = 00H ベリファイを行わない。
 01H ベリファイを行う。
 DL = 00H

リターン なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 AL には、01H (各ディスクへの書き込みごとに、そのベリファイを行う) または 00H (ベリファイなしで書き込みを行う) のいずれかを、また DL には 00H をセットします。MS-DOS では、ディスクに書き込みが行われるごとに、このフラグの検証を行います。

フラグは通常、オフになっていますが、重要なデータをディスクに書き込む場合、このフラグをオンにした方が望ましいでしょう。ただし、ディスクエラーはまれであり、ベリファイを行った場合は処理速度が遅くなるので、通常のデータを処理するときは、オフにしておいてもよいでしょう。

マクロ定義

```

verify macro switch
    mov     al, switch
    mov     ah, 2EH
    mov     dl, 0
    int     21H
endm
  
```

例

つぎのプログラムは、ドライブ A にある片面ディスクの内容をドライブ B にあるディスクにコピーし、書き込みが行われるごとにベリファイを行うものです。このプログラムは、32K バイトのバッファを使用します。

```

on          equ 1
off         equ 0
;
prompt      db  "Source in A,  target in B", 13, 10
            db  "Any key to start.  $"
start       dw
buffer      db  64 dup (512 dup (?))    ; 64 セクタ
;
func_2EH:   display    prompt            ; prompt をスクリーンに出力(09H)
            read_kbd    ; キーボード入力(08H)
            verify     on                ; ベリファイフラグを on にセット
            mov        cx, 5             ; 64 セクタを 5 回コピーする
copy:       push       cx                ; カウンタのセーブ
            abs_disk_read 0, buffer, 64, start
                                           ; アブソリュートディスクリード (INT25H)
            abs_disk_write 1, buffer, 64, start
                                           ; アブソリュートディスクライト (INT26H)
            add        start, 64         ; 次の 64 セクタの処理
            pop        cx                ; カウンタをリストア
            loop       copy              ; コピー処理を継続
            verify     off               ; ベリファイフラグを off にセット

```


INT 21H

Get Disk Transfer Address

ファンクション 2FH

機能	ディスク転送アドレスを得る
コール	AH = 2FH
リターン	ES : BX ディスク転送アドレス

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSh		FLAGSL
CS		
DS		
SS		
ES		

解説 ディスク転送アドレスのオフセットをBXに、ESにセグメントを返します。

エラーリターン なし

マクロ定義

```

get_dta          macro
                  mov     ah, 2FH
                  int     21H
                  endm

```

例 つぎのプログラムは、カレントディスクの転送アドレスを表示するプログラムです。

```

message          db      "DTA --- : ", 0DH, 0AH, "$"
sixteen          db      10H
temp             db      2 dup (?)
;
func_2FH:        get_dta          ; THIS FUNCTION
                  mov     word ptr temp, ES ; ディスク転送アドレスを得る

```

convert	temp[1], sixteen, message[07H]; CONVERT についての
convert	temp, sixteen, message[09H] ; 説明は章末参照
convert	bh, sixteen, message[0CH]
convert	bl, sixteen, message[0EHH]
display	message ; message をスクリーンに出力(09H)

INT 21H

Get MS-DOS Version Number

ファンクション 30H

機能	MS-DOS バージョン番号を得る
コール	AH = 30H
リターン	AL バージョン番号の整数部 AH バージョン番号の小数部 BH OEM のシリアル番号 BL : CX 24 ビットのユーザーシリアル番号 (OEM によって異なる)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説	MS-DOSバージョン番号を返します。このとき、AL、AHにはそれぞれのバージョン番号の整数部、小数部がはいります。すなわち、MS-DOS 3.3 の場合、AL は3に、AHは30(1EH)になります。ALが0の場合は、MS-DOS 2.0以前のバージョンを表します。
-----------	--

PC-9800 シリーズでは、BH には FFH、BL : CX には 000000 H が返されます。

エラーリターン なし

マクロ定義	get-version	macro
	mov	ah, 30H
	int	21H
	endm	

例	つぎのプログラム例は、1.28 以上の MS-DOS のバージョンを表示します。
----------	--

```

message      db          "MS-DOS Version      ",0DH, 0AH, "$"
ten          db          0AH
;
func_30H:    get-version      ; MS-DOS のバージョン番号を得る
              cmp             al, 0      ; 1.28 以上か?

```



```

jng      return      ;いいえの時、処理終了
convert  al, ten, message [0FH] ;はいの時、バージョン番号を
convert  ah, ten, message [12H] ;ASCIIコードに変換し、
                                ;バッファにセット
display  message      ;MS-DOSのバージョンを
                                ;スクリーンに表示(09H)
    
```

INT 21H

Keep Process

ファンクション 31H

機 能	キーププロセス
コール	AH = 31H AL 抜け出しコード DX パラグラフでのメモリサイズ (サイズは16バイト単位)
リターン	なし

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解 説	<p>終了した後、メモリ上にプログラムを残します。また、デバイスの特殊な割り込みハンドルにも使用される場合があります。割り込みタイプ 27H (プログラムをメモリにとどめた終了) と異なり、64K バイト以上のプログラムの常駐を許し、CS (プログラムセグメントプレフィックスのセグメントアドレス) の設定を必要としません。MS-DOS V2.0 との互換性を保つ必要があるような特別な場合を除いて、割り込みタイプ 27H ではなく、ファンクション 31H を使用してください。</p> <p>DX は、プログラムが必要とするメモリのパラグラフ数 (1 パラグラフ = 16 バイト) でなければなりません。AL は、抜け出しコードでなければなりません。</p> <p>EXE 形式のプログラムの場合は、注意が必要です。DX の値は、常駐するプログラムの合計のサイズでなければならず、常駐するプログラムのコードのセグメントのサイズであってはなりません。</p> <p>ユーザーがよく間違える例としては、100H バイトのプログラムヘッダプレフィックスを忘れて、プログラムのコードサイズのみをパラグラフ数で DX にセットしてしまうことがあります。</p> <p>MS-DOS は、現在のプロセスを終了し、イニシャルアロケーションブロックをパラグラフの大きさにセットします。このコールは、このプロセスに属する他のアロケーションブロックを解放するものではありません。AL 内に渡された抜け出しコードは、ファンクション 4DH を通して、親プロセスから取得することが</p>
------------	--

できます。

エラーリターン なし

マクロ定義

```
keep-process macro return-code, last-byte
    mov     al, return-code
    mov     dx, offset last-byte
    mov     cl, 4
    shr     dx, cl
    inc     dx
    mov     ah, 31H
    int     21H
endm
```

例

このコールの使い方の大半は、マシンごとのルーチンに依存しますので、プログラムは省略します。マクロ定義を参照してください。

INT 21H

CTRL-C Check

ファンクション 33H

機能 <CTRL-C> 検査のセット/リセット

コール AH = 33H

AL ファンクション

00H 現在のステータスを得る

01H ステータスのセット

DL (セットする場合: AL = 1)

00H オフ

01H オン

リターン DL = 00H オフ

01H オン

AL = FFH エラー (コールしたときの AL が
00H または 01H でない)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 MS-DOS の <CTRL-C> 検査の状態を得るか、またはセットします。AL の値はつぎのいずれかでなければなりません。

AL=0 DL に現在の <CTRL-C> 検査の状態を返す。

1 DL の値で、<CTRL-C> 検査の状態を設定する。

AL が 0 の場合、DL は現在の <CTRL-C> 検査の状態を返します。AL が 1 の場合、DL の値はセットされる <CTRL-C> 検査の状態です (DL=0: オフ, DL=1: オン)。AL が 0 または 1 でない場合、AL は FFH を返し、<CTRL-C> 検査の状態は影響を受けません。

MS-DOS は通常、01H から 0CH までのファンクションコール動作を実行しているときのみ、<CTRL-C> の検査を行います。<CTRL-C> の検査がオンの場合、すべてのシステムコールについてこの検査を行わせることができます。たとえば、<CTRL-C> の検査がオフの場合、すべてのディスクアクセスは割り込みが実行されることなしに続行しますが、オンの場合、ディスクアクセスを開始させたシステムコールにおいても <CTRL-C> の割り込みが実行されます。

注意：

ファンクションコール 06 H, 07 H によって、データとして<CTRL-C>を読み取るプログラムは、<CTRL-C>検査がオフであることを確認する必要があります。

エラーリターン

AL

FFH = AL に渡されたファンクションが 0~1 の範囲外でした。

マクロ定義

```
ctrl-c-ck      macro      action, state
                mov        al, action
                mov        dl, state
                mov        ah, 33H
                int        21H
                endm
```

例

つぎのプログラムは、<CTRL-C>検査がオンかオフかのメッセージを表示します。

```
message      db      "Control-C checking", "$"
on           db      "on", "$", 0DH, 0AH, "$"
off          db      "off", "$", 0DH, 0AH, "$"
;
func_33H:    display  message      ; message を表示(09H)
             ctrl-c-ck  0          ; (CTRL-C)検査
             cmp        dl, 0      ; オフか?
             jg         ck_on      ; いいえの時, ck_on へ
             display    off        ; はいの時, "off"をスクリーンに
                                     ; 出力(09H)
             jmp        return     ; 処理終了
ck_on:       display    on         ; "on"をスクリーンに出力(09H)
```

INT 21H

Get Interrupt Vector

ファンクション 35H

機能 割り込みベクタを得る

コール AH = 35H
AL 割り込み番号

リターン ES:BX 割り込みルーチンの位置を示す。

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 指定した割り込みの、割り込みベクタのアドレスを得ます。AL は、割り込み番号でなければなりません。BX には、割り込みルーチンのオフセットアドレス (ES はセグメントアドレス) が返されます。

互換性の問題が生じないために、割り込みベクタをメモリに直接読み書きしないでください。MS-DOS V2.0 との互換性を保つ必要があるような特別な場合を除いて、割り込みベクタを得るにはファンクション 35H を、割り込みベクタのセットにはファンクション 25H (割り込みベクタのセット) を使用してください。

マクロ定義

```

get_vector    macro    interrupt
               mov     al, interrupt
               mov     ah, 35H
               int     21H
               endm
  
```


例

つぎのプログラムは、割り込みタイプ 25 H (アブソリュートディスクリード) のアドレス (CS:IP) を表示します。

```

message      db      "Interrupt 25H -- CS: 0000 IP: 0000"
              db      0DH, 0AH, "$"
vec_seg      db      2 dup (?)
vec_off      db      2 dup (?)
;
func_35H:    push     es                ; ES をセーブ
              get_vector 25H            ; 割り込みベクタを得る
              mov      ax, es           ; INT25H のセグメントアドレス
                                              ; を AX にセット
              pop      es              ; ES をリストア
              convert   ax, 16, message[20]; 章末参照
              convert   bx, 16, message[28]; 章末参照
              display   message         ; message をスクリーンに出力(09H)

```

INT 21H

Get Disk Free Space

ファンクション 36H

- 機 能** ディスクのフリースペースを得る
- コール** AH = 36H
DL ドライブ番号 (00H = カレント,
01H = A, 02H = B...)
- リターン** BX 使用可能なクラスタ数
DX 1ドライブ当たりのクラスタ数
CX 1セクタ当たりのバイト数
AX 1クラスタ当たりのセクタ数
= FFFFH ドライブ番号が無効

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSh		FLAGSL
CS		
DS		
SS		
ES		

- 解 説** 指定したドライブの使用可能なクラスタ数、ディスクのメディアの情報（計算によって使用可能なバイト数を得られます）を返します。DL は、ドライブ番号 (00H = カレント, 01H = A ドライブ) でなければなりません。ドライブ番号が無効ならばつぎの情報を返します。

- AX 1クラスタ当たりのセクタ数
BX 使用可能なクラスタ数
CX 1セクタ当たりのバイト数
DX 全クラスタ数 (1ドライブ当たりのクラスタ数)

ドライブ番号が無効な場合、AX は、FFFFH を返します。

ファンクション 1BH, 1CH は、V 2.0 以前の MS-DOS と互換性を保つために用意されています。ファンクション 1BH, 1CH の代わりに、このコールを使用してください。

エラーリターン

AX

FFFFH = DL で指定されたドライブ番号は、無効でした。

```

マクロ定義   get-disk-space  macro  drive
                                mov    dl, drive
                                mov    ah, 36H
                                int     21H
                                endm

```

例 つぎのプログラムは、ドライブ B のディスクのフリースペース情報を表示します。

```

message      db    "clusters on drive B.", 0DH, 0AH      ; DX
              db    "clusters available,", 0DH, 0AH      ; BX
              db    "sectors per cluster.", 0DH, 0AH      ; AX
              db    "bytes per sector,", 0DH, 0AH, "$"    ; CX
;
func_36H:    get-disk-space  2                          ; ディスクのフリースペースを得る
              convert      ax, 10, message[55]; 章末参照
              convert      bx, 10, message[28]; 章末参照
              convert      cx, 10, message[83]; 章末参照
              convert      dx, 10, message      ; 章末参照
              display      message              ; message をスクリーンに出力(09H)

```


INT 21H

Get Country Data

ファンクション 38H

機能 国別情報を得る

コール AH = 38H

AL = 00H 現在の国

01H USA 規格

51H 日本規格

DS: DX 32 バイトのメモリ領域に対する
ポインタ

リターン キャリーフラグがセットされた場合

AX = 02H 無効なファンクション

キャリーフラグがセットされない場合

DS: DX に、国についてのデータがセットさ
れます。

解説

このファンクション 38H は、MS-DOS がキーボード、スクリーンの制御に使う国別情報を得る、またはセットします (Set Country Data: 詳細に付いては、つぎのファンクションコールを参照してください)。DX は、国別情報についての 32 バイトのメモリ領域のオフセットアドレス (セグメントアドレスは、DS) でなければなりません。AL はカントリーコードで、その内容をつぎに示します。

AL 意味

0 現在の国についての情報を得る。

1~FEH このコードで指定された国についての情報を得る。

DS: DX でアドレスを指定された 32 バイトのメモリ領域の内容をつぎに示します。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

00H	2 バイト	日付表示フォーマット
02H	5 バイト	ASCIZ ストリング 通貨記号
07H	2 バイト	ASCIZ ストリング 3 桁ごとの区切り記号
09H	2 バイト	ASCIZ ストリング 10進分離記号
0BH	2 バイト	ASCIZ ストリング 日付分離記号
0DH	2 バイト	ASCIZ ストリング 時刻分離記号
0FH	1 バイト	ビット・フィールド
10H	1 バイト	通貨桁
11H	1 バイト	時刻フォーマット
12H	4 バイト	ケースマッピングコール
16H	2 バイト	ASCIZ ストリング データ・リスト分離記号
18H		

これらのエントリの大部分のフォーマットは、ASCIZ (NUL コードで終了する ASCII ストリング) ですが、テーブルの索引を簡単にするために、各フィールドは固定された大きさが割り当てられます。

日付の項目には以下のフォーマットで値が入ります。

- 0 USA 規格 m/d/y
- 1 ヨーロッパ規格 d/m/y
- 2 日本規格 y/m/d

ビットフィールドには、8 ビットの値が入ります。現在定義されていないすべてのビットは、ランダムな値を持っていると想定しなければなりません。

- 0 ビット目=0 通貨記号が金額の前に付く場合。
- =1 通貨記号が金額の後に付く場合。
- 1 ビット目=0 通貨記号が金額の直前に付く場合。
- =1 通貨記号と金額の間に、スペースを入れる場合。

時刻フォーマットは、以下の値が入ります。

- 0 12 時制
- 1 24 時制

通貨桁フィールドは、通貨金額の小数点以下の桁数を示します。

ケースマッピングコールとは、FAR 手続きのことで、これによって 80H から FFH までの文字に対して国に固有の小文字から大文字のマッピングが行われます。AL に入っているマップすべき文字を使用して、このコールが行われます。AL 内に文字が入っている場合、この文字の正しい大文字コードが返されます。変更されるレジスタは AL および FLAGS のみです。このルーチンに 30H 未満のコードを渡すことは可能ですが、この範囲の文字に対しては、何も行われません。この場合、マッピングは行われず、AL は変更されません。

エラーリターン

AL

02H = AL に渡された国が、見つかりませんでした(指定された国のためのテーブルが、存在しません。)

マクロ定義	get-country	macro	country, buffer
		local	gc-01
		mov	dx, offset buffer
		mov	ax, country
	gc-01H:	mov	ah, 38H
		int	21H
		endm	

例

つぎのプログラムは、時刻と日付を現在のカントリーコードで表示し、通貨記号と区切り記号を使って、999, 999 と 99/100 を表示します。

```
time      db      " : : ", 5 dup (20H), "$"
date      db      " / / ", 5 dup (20H), "$"
number    db      "999?999?99", 0DH, 0AH, "$"
data-area db      32 dup (?)
;
```



```

func_38H:  get_country    0, data_area    ; 国別情報を得る
            get_time      ; 時刻を得る(2CH)
            byte_to_dec   ch,  time      ; 変換に関するマクロの説明は
            byte_to_dec   cl,  time[03H] ; 章末を参照
            byte_to_dec   dh,  time[06H]
            get_date      ; 日付を得る(2AH)
            sub           cx,  1900      ; 下2桁を得る
            byte_to_dec   cl,  date[06H] ; 章末参照
            cmp           word ptr data_area, 0 ; カントリーコードをチェック
            jne           not_usa        ; USA でない時, not_usa へ
            byte_to_dec   dh,  date      ; 章末参照
            byte_to_dec   dl,  date[03H] ; 章末参照
            jmp           all_done
not_usa:    byte_to_dec   dl,  date      ; 章末参照
            byte_to_dec   dh,  date[03H] ; 章末参照
all_done:   mov           al, data_area[07H] ; NUMBER に3桁ごとの区切
            mov           number[03H], al  ; り記号を入れる
            mov           al, data_area[09H] ; AMOUNT に10進分離記号を
            mov           number[07H], al  ; 入れる
            display       time             ; time をスクリーンに出力(09H)
            display       date             ; date をスクリーンに出力(09H)
            display_char  data_area[02H]   ; 文字をスクリーンに出力(02H)
            display       number           ; number をスクリーンに出力(09H)

```

INT 21H

Set Country Data

ファンクション 38H

機能 国別情報をセットする

コール AH = 38H

DX = FFFFH

AL = FFH 以外 カントリーコード

FFH BX にカントリーコードが入っている

BX (AL=FFH の場合)

FFH 以上のカントリーコード

リターン キャリーフラグがセットされた場合

AX = 02H 無効なカントリーコード

キャリーフラグがセットされない場合

エラーなし

解説

このファンクション 38 H は、MS-DOS がキーボード、スクリーン等の制御に使う国別情報をセット、または、国別情報を取得します (Get Country Data : 詳細については、先のファンクションコールを参照してください)。DX は、FFFFH (−1) でなければなりません。AL はカントリーコードで、その内容をつぎに示します。

AL 意味

01H~FEH このコードで指定された国のカントリーコード。

FFH BX で指定された国のカントリーコード。

カントリーコードは、通常その国の国際電話プレフィクスコードです。
(PC-9800 シリーズでは AL=01 H (USA 規格)、AL=51 H (日本規格) のみ指定できます)

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

02H=AL に渡された国が、見つかりませんでした (指定された国のためのテーブルが、存在しません)。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

マクロ定義	<pre> set-country local mov dx, 0FFFFH mov ax, country cmp ax, 0FFH jl sc_01 mov bx, country mov al, 0FFH mov ah, 38H int 21H endm </pre>	<pre> country sc_01: </pre>
-------	--	---------------------------------------

例 つぎのプログラムは、カントリーコードをイギリス(44)に変えます。

```

uk      equ      44
;
func_38H: set-country uk      ; 国別情報のカントリーコードをイギリスにセット
          jc      error
          
```


INT 21H

Create Directory

ファンクション 39H

機能 ディレクトリの作成

コール AH = 39H
DS: DX パス名の位置リターン キャリーフラグがセットされた場合
AX = 03H 無効なパス
05H アクセスの否定
キャリーフラグがセットされない場合
エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 新しいサブディレクトリを作成します。DX は、新しいサブディレクトリのパス名を表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) でなければなりません。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

03H = 指定されたパスが無効であったか、または存在しませんでした。

05H = 親ディレクトリ内に空きのスペースがないか、すでに同名のディレクトリ/ファイルが存在しているので、ディレクトリを作成することが、できませんでした。

マクロ定義	make_dir	macro	path
		mov	dx, offset path
		mov	ah, 39H
		int	21H
		endm	

例

つぎのプログラムは、ドライブBのディスク上のルートディレクトリ下に、“NEWDIR” という名のサブディレクトリを作成し、カレントディレクトリを一度、“NEWDIR” に移し、つぎにもとのディレクトリに戻り、“NEWDIR” を削除します。また、移動するごとに、カレントディレクトリを表示します。

```

old_path    db    "b: ¥", 0, 63 dup (?)
new_path    db    "b: ¥newdir", 0
buffer      db    "b: ¥", 0, 63 dup (?)
;
func_39H:   get_dir    2, old_path[03H]    ; カレントディレクトリ情報を得る
            jc         error_get
            display_asciz old_path        ; 章末参照
            make_dir   new_path          ; ディレクトリ NEWDIR を作成
            jc         error_make
            change_dir  new_path          ; カレントディレクトリを NEWDIR
            jc         error_change      ; に変換(3BH)
            get_dir    2, buffer[03H]    ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer        ; 章末参照
            change_dir  old_path          ; カレントディレクトリの変更(3BH)
            jc         error_change
            rem_dir     new_path          ; ディレクトリ NEWDIR を削除(3AH)
            jc         error_rem
            get_dir    2, buffer[03H]    ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer        ; 章末参照

```

INT 21H

Remove Directory

ファンクション 3AH

機能 ディレクトリの削除

コール AH = 3AH
 DS: DX パス名の位置

リターン キャリーフラグがセットされた場合
 AX = 03H 無効なパス
 05H アクセスの否定
 10H カレントディレクトリ
 キャリーフラグがセットされない場合
 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 サブディレクトリを削除します。DX は、削除されるサブディレクトリのパス名を表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) です。削除されるディレクトリは空 (ファイル、ディレクトリを含んでいない) でなければなりません。また、カレントディレクトリを削除することはできません。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

03H = 指定されたパスが無効であったか、または存在しません。

05H = 指定されたパスが空でなかったか、ディレクトリでないか、ルートディレクトリであるか、その他無効な情報が入っていました。

10H = 指定されたディレクトリは、カレントディレクトリでした。

マクロ定義	rem_dir	macro	path
		mov	dx, offset path
		mov	ah, 3AH
		int	21H
		endm	

例 つぎのプログラムは、ドライブBのディスク上のルートディレクトリ下に、“NEWDIR” という名のサブディレクトリを作成し、カレントディレクトリを一度、“NEWDIR” に移し、つぎにもとのディレクトリに戻り、“NEWDIR” を削除します。また、移動するごとに、カレントディレクトリを表示します。

```

old_path    db    "b: ¥", 0, 63 dup (?)
new_path    db    "b: ¥newdir", 0
buffer      db    "b: ¥", 0, 63 dup (?)
;
func_3AH:   get_dir    2, old_path[03H]    ; カレントディレクトリ情報を得る(47H)
            jc         error_get
            display_asciz old_path        ; 章末参照
            make_dir   new_path          ; ディレクトリ NEWDIR を作成(39H)
            jc         error_make
            change_dir  new_path          ; カレントディレクトリを NEWDIR
                                           ; に変換(3BH)
            jc         error_change
            get_dir    2, buffer[03H]     ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer          ; 章末参照
            change_dir  old_path          ; カレントディレクトリの変更(3BH)
            jc         error_change
            rem_dir     new_path          ; ディレクトリ NEWDIR を削除(3AH)
            jc         error_rem
            get_dir    2, buffer[03H]     ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer          ; 章末参照

```

INT 21H

Change Current Directory

ファンクション 3BH

機能	カレントディレクトリの変更
コール	AH = 3BH DS:DX パス名の位置
リターン	キャリーがセットされた場合 AX = 03H 無効なパス キャリーがセットされない場合 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解説	<p>カレントディレクトリを変更します。DX は、新しいサブディレクトリのパス名を表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) でなければなりません。ディレクトリを指定する文字列は 64 文字以内です。</p> <p>指定されたパス名のディレクトリが存在しない場合、カレントディレクトリは変更されません。エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。</p>
-----------	--

エラーリターン

AX

03H = DS:DX で指定されたパスがファイルを示していたか、またはパスが無効でした。

マクロ定義	change_dir	macro	path
		mov	dx, offset path
		mov	ah, 3BH
		int	21H
		endm	

例

つぎのプログラムは、ドライブBのディスク上のルートディレクトリ下に、“NEWDIR” という名のサブディレクトリを作成し、カレントディレクトリを一度、“NEWDIR” に移し、つぎにもとのディレクトリに戻り、“NEWDIR” を削除します。また、移動するごとに、カレントディレクトリを表示します。

```

old_path    db        "b: ¥", 0, 63 dup (?)
new_path    db        "b: ¥newdir ", 0
buffer      db        "b: ¥", 0, 63 dup (?)
;
func_3BH:   get_dir    2, old_path[03H]    ; カレントディレクトリ情報を得る(47H)
            jc         error_get
            display_asciz old_path        ; 章末参照
            make_dir   new_path          ; ディレクトリ NEWDIR を作成(39H)
            jc         error_make
            change_dir new_path          ; カレントディレクトリの変更
            jc         error_change
            get_dir    2, buffer[03H]    ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer        ; 章末参照
            change_dir old_path          ; カレントディレクトリの変更
            jc         error_change
            rem_dir    new_path          ; ディレクトリを削除(3AH)
            jc         error_rem
            get_dir    2, buffer[03H]    ; カレントディレクトリを得る(47H)
            jc         error_get
            display_asciz buffer        ; 章末参照

```


INT 21H

Create Handle

ファンクション 3CH

機能	ハンドルの作成
コール	AH = 3CH DS: DX パス名の位置 CX ファイルの属性
リターン	キャリーがセットされた場合 AX = 03H 無効なパス 04H オープンされているファイルが多すぎる。 05H アクセスの否定 キャリーがセットされない場合 AX ファイルハンドル

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 ファイルを作成し、最初の利用可能なハンドルを割り当てます。DX は、新しいファイルのパス名を表す ASCIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。CX は、ファイルに割り当てられた属性でなければなりません。ファイルの属性については、1.5.6 を参照してください。

同名のファイルが存在しない場合、新規のファイルを作成します。同名のファイルがある場合は、そのファイルの大きさを 0 にします。CX 内の属性はファイルに割り当てられ、読み出し／書き込みのためにオープンされます。AX は、ファイルハンドルを返します。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

03H = 指定したパスが無効でした。

04H = 指定された属性のファイルが作成されましたが、リード／ライトアクセスをするためのハンドル、または内部システムテーブルに空きスペースがありませんでした。

05H = CX で指定された属性に作成不可能なもの(ディレクトリ, ボリュームラベル)が入っていたか, ファイルを保護する属性がすでに与えられていた, またはディレクトリに同じ名前のファイルが存在していました。

マクロ定義

```
create_handle macro      path, attrib
                        mov      dx, offset path
                        mov      cx, attrib
                        mov      ah, 3CH
                        int      21H
endm
```

例

つぎのプログラムは, ドライブ B のディスクに "DIR. TMP" という名前のファイルを作成します。このファイルは, カレントディレクトリにある各ファイルのファイル名と拡張子を含んでいます。

```
srch_file    db          "b: *. *", 0
tmp_file     db          "b: dir. tmp", 0
buffer       db          43 dup (?)
handle       dw          ?
;
func_3CH:    set_dta      buffer          ; ディスク転送アドレスのセット(1AH)
             find_first_file srch_file, 16H ; 最初に一致するファイル名の検索(4EH)
             cmp          ax, 12H          ; これ以上ファイルがないか?
             je           all_done         ; はいの時, all_done へ
             create_handle tmp_file, 0     ; ハンドルの作成
             jc           error
             mov          handle, ax       ; ハンドルのセーブ
write_it:    write_handle handle, buffer[1EH], 12 ; ファイルに書き込む(40H)
             find_next_file                ; 次に一致するファイル名の検索(4FH)
             cmp          ax, 12H          ; 他のエントリは存在するか?
             je           all_done         ; いいえの時, all_done へ
             jmp          write_it         ; はいの時, レコードを書き込む
all_done:    close_handle handle          ; ハンドルのクローズ(3EH)
```

INT 21H

Open Handle

ファンクション 3DH

機能 ハンドルのオープン

コール AH = 3DH
AL ファイルアクセスコントロール
DS: DX パス名の位置

リターン キャリーフラグがセットされた場合
AX = 01H 無効なファンクションコード
02H ファイルが存在しない
03H 無効なパス
04H オープンされているファイルが多すぎる。
05H アクセスの否定
0CH 無効なアクセス

キャリーがセットされない場合

AX ファイルハンドル

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 このファンクションは、システムファイル、隠されたファイルも含めたあらゆるファイルを、入力、または出力のモードでオープンします。DX は、オープンされるファイルのパス名を表す ASCIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。AL は、ファイルをオープンする方法を表すコード（ファイルアクセスコントロールを参照してください）でなければなりません。

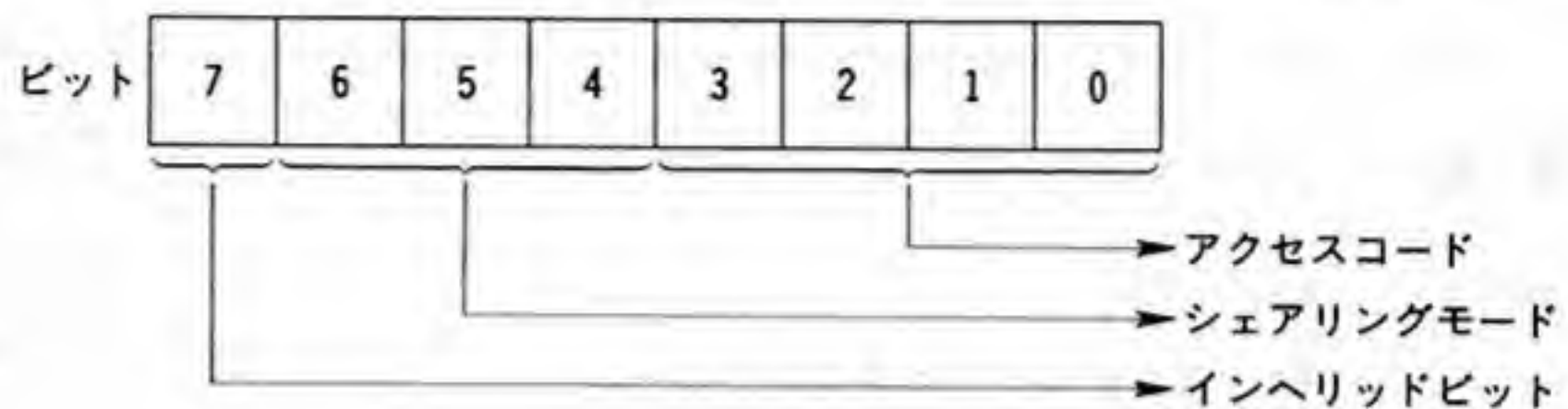
エラーがなかった場合、MS-DOS は、ハンドルの最初の 1 バイトのリード／ライトの設定をセットします。

ファイルアクセスコントロール

AL に入れるコードは、つぎの 3 つのコードの集まりです。

1. ファイルをオープンするモードが、リード、ライト、リード／ライトのいずれかであるかを表します（アクセスコード：0～3 ビット）。
2. 他のプロセスがファイルをアクセスできるかどうかを表します（シェアリングモード：4～6 ビット）。

3. ファイルが子プロセスによって受け渡されるかどうかを表します (インヘリッドビット: 7ビット)



インヘリッドビット

この最上位のビットは、ファイルが、ファンクション 4BH, コード 00H (プログラムのロードと実行) によって作成された子プロセスから、情報を受け渡されるかどうかを表しています。0 の場合、受け渡されます。1 の場合は、受け渡されません。

シェアリングモード

シェアリングモード (6~4ビット) は、他のプロセスが、オープンしているファイルをアクセスできるかどうかを表しています。

ビット 6~4	シェアリングモード	意味
000	コンパチブル	どんなプロセスでも、このモードでは何度でもオープンできます。他のシェアリングモードでオープンできません。
001	リード/ライト不可	どんなプロセス (カレントプロセス自身さえも) も、コンパチブルモードでのオープン、読み出し、または書き込みのためのアクセスはできません。
010	ライト不可	他のプロセスは、コンパチブルモードでのオープン、書き込みのためのアクセスはできません。
011	リード不可	他のプロセスは、コンパチブルモードでのオープン、読み出しのためのアクセスはできません。
100	不可なし	他のプロセスは、コンパチブルモードでのオープンはできません。

アクセスコード

アクセスコード (AL の 3~0 ビット) は、ファイルがどのようにアクセスできるかを表しています。

ビット 3~0	許されるアクセス	意味
0000	リード	リード不可, リード/ライト不可のシェアリングモードでオープンできません。
0001	ライト	ライト不可, リード/ライト不可のシェアリングモードでオープンできません。
0010	リード/ライト	リード不可, ライト不可, リード/ライト不可のシェアリングモードでオープンできません。

エラーが起きた場合、キャリーフラグがセットされ、AX に、エラーコードが返されます。

エラーリターン

AX

- 01H = ファイルシェアリングは、同一のシェアリングモード (AL の 4~6 ビット) でロードされなければなりません。
- 02H = ファイル名が無効か、または存在しません。
- 03H = パス名が無効か、または存在しません。
- 04H = カレントプロセス中に利用できるハンドルがないか、または内部システムテーブル一杯。
- 05H = プログラムがディレクトリかボリューム ID をオープンしようとしたか、またはリードオンリーのファイルに書き込もうとした。
- 0CH = アクセスコードが 0, 1, 2 のいずれでもない。

ファイルシェアリングによるエラーのために、システムコールが失敗した場合、MS-DOS は割り込みタイプ 24H、エラーコード 02H (ドライブの準備ができていない) を実行します。続いて起こるファンクション 59H (拡張されたエラーを得る) は、シェアリングの破壊を表す拡張されたエラーコードを返します。

ファイルをオープンしたとき、このファイルで他のプロセスが実行しうる、あらゆる操作についての情報を、MS-DOS に与えておくことが重要です (シェアリングモード)。デフォルトのシェアリングモード (コンパチブルモード) は、ファイルへの他のプロセスのアクセスをすべて否定します。あるプロセスがファイルを扱っているときに、他のプロセスがそのファイルを読み込むのを認める場合、ビット 5 をセットし、他のプロセスへの読み込みを許します。

同様に、カレントプロセスが実行するであろう操作を明確にすることも重要です（アクセスコード）。

デフォルトのアクセスコード（リード／ライト）では、すでにリード不可、ライト不可、リード／ライト不可のいずれかのシェアリングモードでオープンすることはできません。また、あるファイルを読み込むだけの場合、他のすべてのプロセスが、リード不可、リード／ライト不可のどちらかでなければオープンできます。

マクロ定義	open_handle	macro	path, access
		mov	dx, offset path
		mov	al, access
		mov	ah, 3DH
		int	21H
		endm	

例	つぎのプログラムは、ドライブBの“TEXTFILE.ASC”という名のファイルをプリンタに印字します。
---	---

file	db	"b: textfile. asc", 0	
buffer	db	?	
handle	dw	?	
;			
func_3DH:	open_handle	file, 0	; ハンドルのオープン
	mov	handle, ax	; ハンドルのセーブ
read_char:	read_handle	handle, buffer, 1	; 1文字読み込む
	jc	error_read	
	cmp	ax, 0	; ファイルエンドか?
	je	return	; はいの時、処理終了
	print_char	buffer	; いいえの時、文字をプリンタに
			; 出力(05H)
	jmp	read_char	; 次の文字を読み込む

INT 21H

Close Handle

ファンクション 3EH

機 能	ハンドルのクローズ
コール	AH = 3EH BX ファイルハンドル
リターン	キャリーがセットされた場合 AX = 06H 無効なハンドル キャリーがセットされない場合 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 ファンクション 3DH (ハンドルのオープン), または 3CH (ハンドルの作成) でオープンされたファイルをクローズします。BX は、クローズすべきオープンされたファイルハンドルでなければなりません。

エラーがない場合、MS-DOS はファイルをクローズし、すべての内部バッファも解放されます。エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

06H = BX に渡されたハンドルは、現在オープンされていません。

マクロ定義	close_handle	macro	handle
		mov	bx, handle
		mov	ah, 3EH
		int	21H
		endm	

例 つぎのプログラムは、ドライブ B のディスクに "DIR. TMP" という名前のファイルを作成します。このファイルは、カレントディレクトリにあるファイルのファイル名と拡張子を含んでいます。

```

srch_file    db    "b: *. *", 0
tmp_file     db    "b: dir. tmp", 0
buffer       db    43 dup (?)
handle       dw    ?
;
func_3EH:    set_dta    buffer        ; ディスク転送アドレスのセット(1AH)
             find_first_file srch_file, 16H ; 最初に一致するファイル名の検索(4EH)
             cmp        ax, 12H      ; これ以上ファイルがないか?
             je         all_done     ; はいの時, all_done へ
             create_handle tmp_file, 0 ; ハンドルの作成(3CH)
             jc         error_create
             mov        handle, ax    ; ハンドルのセーブ
write_it:     write_handle handle, buffer[1EH], 12 ; ファイルに書き込む(40H)
             jc         error_write
             find_next_file          ; 次に一致するファイル名の検索(4FH)
             cmp        ax, 12H      ; 他のエントリは存在するか?
             je         all_done     ; いいえの時, all_done へ
             jmp        write_it     ; はいの時, レコードを書き込む
all_done:     close_handle handle     ; ハンドルのクローズ(3EH)
             jc         error_close

```

INT 21H

Read Handle

ファンクション 3FH

機能 リードハンドル

コール AH = 3FH

DS: DX バッファの位置

CX 読み込むべきバイト数

BX ファイルハンドル

リターン キャリーがセットされた場合

AX = 05H アクセスできない。

06H ハンドルが無効

キャリーがセットされない場合

AX 読み込まれたバイト数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

ハンドルで指定されたファイル、またはデバイスからデータを読み込みます。BX は、ハンドルでなければなりません。CX は、読み込むバイト数でなければなりません。DX は、バッファのオフセットアドレス (DS は、セグメントアドレス) でなければなりません。

エラーがなかった場合、AX は、読み込まれたバイト数を返します。ファイルの先頭が EOF (ファイルの終りを表すコード) の場合、AX は、0 を返します。CX で指定されたバイト数が、すべてバッファに転送される保証はありません。たとえば、このファンクションを使ってキーボードからデータを読み込む場合、最高1行分 (最初のキャリッジリターンを入力するまで) のデータしか読み込みません。

このファンクションを使って、標準入力から読み込む場合、リダイレクト処理が可能になります。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

05H = BX で渡されたハンドルは、リードが許可されていないモードでした。

06H=BX で渡されたハンドルは、現在オープンされていません。

マクロ定義	read_handle	macro	handle, buffer, bytes
		mov	bx, handle
		mov	dx, offset buffer
		mov	cx, bytes
		mov	ah, 3FH
		int	21H
		endm	

例 つぎのプログラムは、ドライブBのディスク上の“TEXTFILE.ASC”という名前のファイルを表示します。

```

filename      db      "b: ¥ textfile . asc", 0
buffer        db      129 dup (?)
handle        dw      ?
;
func_3FH:     open_handle filename, 0          ; ハンドルのオープン(3DH)
              jc        error_open
              mov       handle, ax             ; ハンドルのセーブ
read_file:    read_handle buffer, file_handle, 128
              jc        error_open
              cmp       ax, 0                  ; ファイルエンドか?
              je        return                 ; はいの時, 処理終了
              mov       bx, ax                 ; 読み込んだバイト数を BX にセット
              mov       buffer[bx], "$"       ; 表示するストリングを作成
              display   buffer                 ; buffer をスクリーンに出力(09H)
              jmp       read_file              ; 続けて読み込む

```

INT 21H

Write Handle

ファンクション 40H

機能 ライトハンドル

コール AH = 40H

DS:DX バッファの位置

CX 書き込むべきバイト数

BX ファイルハンドル

リターン キャリーがセットされた場合

AX = 05H アクセスの否定

06H 無効なハンドル

キャリーがセットされない場合

AX 書き込まれたバイト数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 ハンドルで指定されたファイル、またはデバイスにデータを書き込みます。BX は、ハンドルでなければなりません。CX は、書き込むバイト数でなければなりません。DX は、書き込まれるデータのオフセットアドレス (DS は、セグメントアドレス) でなければなりません。

エラーがなかった場合、AX は、書き込まれたバイト数を返します。ディスクにファイルを書き込んだ後は必ず、AX をチェックしてください。AX が 0 の場合は、ディスクに書き込む余裕がないことを表します。このコールが実行された後で AX の値が CX で指定された値より少ない場合、キャリーフラグはセットされませんが、エラーであることを表します。

標準出力に書き込んだ場合、出力はリダイレクト可能になります。このファンクションで、CX=0 (バイト数=0) を指定した場合、ファイルサイズは現在のリード/ライトポインタの値にセットされます。アロケーションユニットは、新しいファイルのサイズを満たすように割り付け、または解放されます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

05H = ハンドルは、ライトを許可するモードではありませんでした。

06H = BX で渡されたハンドルは、現在オープンされていません。

マクロ定義

```
write_handle macro    handle, data, bytes
                    mov    bx, handle
                    mov    dx, offset data
                    mov    cx, bytes
                    mov    ah, 40H
                    int     21H
                    endm
```

例

つぎのプログラムは、ドライブ B のディスクに "DIR. TMP" という名前のファイルを作成します。このファイルは、カレントディレクトリにある各ファイルのファイル名と拡張子を含んでいます。

```
srch_file    db    "b: *. *", 0
tmp_file     db    "b: dir. tmp", 0
buffer       db    43 dup (?)
handle       dw    ?

;
func_40H:    set_dta buffer                ; ディスク転送アドレスのセット(1AH)
             find_first_file srch_file, 16H ; 最初に一致するファイル名の検索(4EH)
             cmp    ax, 12H                ; これ以上ファイルがないか?
             je     return                 ; はいの時、処理終了
             create_handle tmp_file, 0     ; ハンドルの作成(3CH)
             jc     error_create
             mov    handle, ax             ; ハンドルのセーブ
write_it:    write_handle handle, buffer[1EH], 12 ; ファイルに書き込む
             jc     error_write
             find_next_file                ; 次に一致するファイル名の検索(4FH)
             cmp    ax, 12H                ; 他のエントリは存在するか?
             je     all_done               ; いいえの時、処理終了
             jmp    write_it              ; はいの時、レコードを書き込む
all_done:    close_handle handle           ; ハンドルのクローズ(3EH)
             jc     error_close
```


INT 21H

Delete Directory Entry

ファンクション 41H

機 能	ディレクトリエントリの削除
コール	AH = 41H DS:DX パス名の位置
リターン	キャリーがセットされた場合 AX = 02H 無効なファイル 05H アクセスの否定 キャリーがセットされない場合 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解 説 ディレクトリエントリを削除することによって、ファイルを削除します。DX は、削除するファイルのパス名を表す ASCIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。ワイルドカード文字は使用できません。

ファイルが実在し、読み出し専用のファイルでなければ、ファイルを削除します。エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

02H = 指定されたパスが無効であったか、または存在しませんでした。

05H = 指定されたパスがディレクトリまたはリードオンリーのファイルでした。

アトリビュートが読み出し専用のファイルを削除する場合は、ファンクション 43H（アトリビュート（属性）の変更）でアトリビュートを変更してください。

マクロ定義

```

delete_entry      macro      path
                   mov       dx, offset path
                   mov       ah, 41H
                   int       21H
                   endm

```

例

つぎのプログラムは、ドライブBのディスク上の1981年12月31日以前の日付のファイルをすべて消去します。

```

year      dw      1981
month     db      12
day       db      31
files     db      ?
message   db      "NO FILES DELETED.", 0DH, 0AH, "$"
path      db      "b: *. *", 0
buffer    db      43 dup (?)
;
func_41H:  set_dta   buffer      ; ディスク転送アドレスのセット(1AH)
           select_disk "B"      ; ドライブBを選択(0EH)
           find_first_file path, 0 ; 最初に一致するファイル名の検索(4EH)
           jnc      compare     ; 一致するファイルを得る
           jmp      all_done     ; 一致しない時, all_doneへ
compare:   convert_date buffer[-1] ; 章末参照
           cmp      cx, year     ; 年は1981より大きい?
           jg       next        ; はいの時, ファイルを削除しない
           cmp      dl, month    ; 12月を越えている?
           jg       next        ; はいの時, ファイルを削除しない
           cmp      dh, day      ; 31日以上?
           jge      next        ; はいの時, 削除しない
           delete_entry bffer[1EH] ; ディレクトリエントリの削除
           jc       error_delete
           inc      files        ; ファイルカウンタをインクリメント
next:      find_next_file      ; 次に一致するファイルの検索
           jnc      compare     ; 日付チェック処理を継続
how_many:  cmp      files, 0     ; これ以上ファイルがない?
           je       all_done     ; はいの時, all_doneへ
           convert   files, 10, message ; 章末参照
all_done:  display   message     ; messageをスクリーンに出力(09H)
           select_disk "A"      ; ドライブAを選択(0EH)

```

INT 21H

Move File Pointer

ファンクション 42H

機 能 ファイルポインタの移動

コール AH = 42H
 CX:DX 移動するバイト数
 AL 移動方法 (解説を参照してください。)
 BX ファイルハンドル

リターン キャリーがセットされた場合
 AX = 01H 無効なファンクション
 06H 無効なハンドル
 キャリーがセットされない場合
 DX:AX 新規のポインタロケーション

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 指定されたハンドルのファイルのリード/ライトポインタを移動します。BX は、ハンドルでなければなりません。CX:DX は、32 ビットのオフセット (CX が上位バイトを含みます) でなければなりません。AL は、ポインタの移動方法で、つぎの値で指定します。

値	機 能
00H	ポインタは、ファイルの先頭からオフセットの位置に移動します。
01H	ポインタは、現在のロケーション (アドレス) とオフセットを加算した位置に移動します。
02H	ポインタは、ファイルの終わりにオフセットを加算した位置に移動します。

DX:AX は、新規のリード/ライトポインタロケーション (32 ビット整数: DX が上位バイトを含みます) を返します。CX:DX を 0, AL を 2 にして、このファンクションをコールし、ファイルの大きさを設定できます。このとき、DX:AX は、ファイルの大きさ (ファイルの最後のバイトのつぎのバイトのオフセット) をバイトで返します。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

01H=AL に渡されたファンクションは、0~2 の範囲外でした。

06H=BX に渡されたハンドルは、現在オープンされていません。

マクロ定義	move_ptr	macro	handle, high, low, method
		mov	bx, handle
		mov	cx, high
		mov	dx, low
		mov	al, method
		mov	ah, 42H
		int	21H
		endm	

例 つぎのプログラムは、1文字の入力を要求し、それを対応する数字に変換 (A=01H, B=02H ……) します。つぎにその数字番目のレコード内容をファイルから読み込み、表示します。そのファイルは、ドライブBのカレントディレクトリ中の“ALPHABET.DAT”で、28バイトの26レコードからなります。

```

file          db          "b: alphabet. dat", 0
buffer        db          28 dup (?), "$"
prompt        db          "Enter letter: $"
crlf          db          0DH, 0AH, "$"
handle        db          ?
record_length dw          28
;
func_42H:     open_handle file, 0          ; ハンドルのオープン(3DH)
              jc          error_open
              mov         handle, ax       ; ハンドルをセーブ
get_char:     display     prompt          ; prompt をスクリーンに出力(09H)
              read_kbd_and_echo          ; 1文字の入力待ち(01H)
              sub         al, 41h         ; 入力文字をレコード番号に変換
              mul         byte ptr record_length ; オフセットを算出
              move_ptr    handle, 0, ax, 0 ; ファイルポインタを移動
              jc          error_move
read_handle   handle, buffer, record_length

```

jc	error_read	
cmp	ax, 0	; ファイルエンドか?
je	return	; はいの時, 処理終了
display	crlf	; crlf をスクリーンに出力(09H)
display	buffer	; buffer をスクリーンに出力(09H)
display	crlf	; crlf をスクリーンに出力(09H)
jmp	get_char	; 次の文字を得る

INT 21H

Get/Set File Attributes

ファンクション 43H

機能 アトリビュート（属性）を得る／セットする

コール AH = 43H

DS:DX パス名の位置

CX (AL = 01 の場合) セットすべき属性

AL ファンクション

00H ファイルの現在の属性を返す。

01H CX で指定された属性のセット

リターン キャリーがセットされた場合

AX = 01H 無効なファンクション

02H 無効なファイル名

03H 無効なパス

05H アクセスの否定

キャリーがセットされない場合

CX 属性 (AL = 00H の場合)

解説

ファイルのアトリビュート（属性）を得る、またはセットします。DX は、ファイルのパス名を表す ASCIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。AL は、アトリビュート（属性）を得るかセットするかを決めるパラメータ（0：アトリビュートを得る、1：アトリビュートをセットする）でなければなりません。

AL が 0 の場合（アトリビュートを得る）、アトリビュートを表す 1 バイトが CX に返されます。AL が 1 の場合（アトリビュートをセットする）、CX は、セットされるアトリビュートでなければなりません。アトリビュートについては 1.5.6 を参照してください。

このファンクションを使って、アトリビュートのボリューム ID ビット (08H)、またはディレクトリビット (10H) を変更することはできません。

エラーが起きた場合、キャリーフラグがセットされ、AX にはエラーコードが返されます。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

エラーリターン

AX

01H=AL に渡されたファンクションは 0~1 の範囲外でした。

02H=指定されたファイル名は無効でした。

03H=指定されたパス名は、無効でした。

05H=CX で指定された属性に、変更できないものが含まれていました(ディレクトリ、ボリュームラベル)。

マクロ定義

```
change_attr      macro      path, action, attrib
                  mov       dx, offset path
                  mov       al, action
                  mov       cx, attrib
                  mov       ah, 43H
                  int       21H
                  endm
```

例

つぎのプログラムは、ドライブ B 上のディスクのカレントディレクトリにある“REPORT. ASM”というファイルのアトリビュートを表示します。

```
header          db          15 dup (20h), "Read-", 0DH, 0AH
                  db          "Filename Only Hidden"
                  db          "System Volume Sub-Dir Archive"
                  db          0DH, 0AH, 0DH, 0AH, "$"
path             db          "b: report. asm", 3 dup (0), "$"
attribute        dw          ?
blanks           db          9 dup (20h), "$"
;
func_43H:        change_attr path, 0, 0      ;アトリビュートを得る
                  jc          error_mode
                  mov         attribute, cx   ;アトリビュートをセーブ
                  display     header          ;header をスクリーンに出力(09H)
                  display     path           ;path をスクリーンに出力(09H)
                  mov         cx, 6          ; (0-5) の 6 ビットをチェック
                  mov         bx, 1
chk_bit:         test         attribute, bx   ;ビットがセットされているか?
                  jz          no_attr        ;いいえの時, no_attr へ
                  display_char "X"           ;はいの時, "X" をスクリーンに出力(02H)
                  jmp short   next_bit       ;次のビット処理へ
```

no_attr:	display_char	20h	; 空白をスクリーンに出力(02H)
next_bit:	display	blanks	; blanks をスクリーンに出力(09H)
	shl	bx, 1	; 次のビットにシフト
	loop	chk_bit	; それをチェック

INT 21H

Get IOCTL Data

ファンクション 44H
コード 00H

機 能	IOCTL データを得る
コール	AH = 44H AL = 00H BX ハンドル
リターン	キャリーフラグがセットされている場合 AX = 01H 無効なファンクション 06H 無効なハンドル キャリーフラグがセットされない場合 DX デバイスデータ

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
	SP	
	BP	
	SI	
	DI	
	IP	
	FLAGSH	FLAGSL
	CS	
	DS	
	SS	
	ES	

解 説	デバイスコントロールデータを得ます。AL は、00H でなければなりません。 BX はハンドルでなければなりません。
------------	---

デバイスデータの2バイトは、DX に返されます。デバイスデータのビット7によってハンドルがファイルを表すかデバイスを表すかが決まり、他のビットの意味も異なります。

・デバイス（ビット7=1）の場合

ビット	値	意味
15		予備
14	1	この装置はファンクション 44 H、コード 02 H (IOCTL キャラクタを送る) と 03 H (IOCTL キャラクタを受け取る) を通して、コントロールストリングを処理できます。このビットは読み出すことはできますが書き込むことはできません。
8~13		予備
7	1	ハンドルはデバイスを表す
6	0	EOF を入力する場合
5	1	コントロールキャラクタをチェックしない
	0	コントロールキャラクタをチェックする
4	1	予備

3	1	クロックデバイス
2	1	NUL デバイス
1	1	コンソール出力
0	1	コンソール入力

ビット 5 がチェックできるコントロールキャラクタは, CTRL-C, CTRL-P, CTRL-S, CTRL-Z で, データとして扱うか, コントロールキャラクタとして扱うかを決めます。ビット 5 をセットして, CTRL-C をデータとして扱う場合, ファンクション 33 H (<CTRL-C>検査のセット/リセット) または MS-DOS の BREAK コマンドで, CTRL-C をチェックしないようにしなければなりません。

・ファイル (ビット 7=0) の場合

ビット 値 意味

15~8 予備

7 0 ハンドルはファイルを表す

6 0 書き込まれたファイル

5~0 デバイス番号 (00H=A, 01H=B……)

エラーが起きた場合, キャリーフラグがセットされ, AX にエラーコードを返します。

エラーリターン

AX

01H=AL が 00H でない

06H=BX のハンドルがオープンされていないか, 無効である。

マクロ定義

```

ioctl_data            macro            code, handle
                      mov            bx, handle
                      mov            al, code
                      mov            ah, 44H
                      int            21H
                      endm

```

例

つぎのプログラムは、標準出力のデバイスデータを得て、コントロールキャラクタをチェックしないようにビット5をセットし、つぎにビット5を0にします。

```

get          equ          0
set          equ          1
stdout       equ          1
;
func_4400H:  ioctl_data    get, stdout      ; IOCTL データを得る
              jc          error
              mov         dh, 0            ; DH をクリア
              or          dl, 20H          ; ビットをセット
              ioctl_data   set, stdout      ; IOCTL データをセット
              jc          error
;
; コントロールキャラクタは、ここではデータとして扱う('raw mode')
;
              ioctl_data   get, stdout      ; IOCTL データを得る
              jc          error
              mov         dh, 0            ; DH をクリア
              and         dl, 0DFH        ; ビット5をクリア
              ioctl_data   set, stdout      ; IOCTL データをセット
;
; コントロールキャラクタは、ここでは処理される('cooked mode')
;

```

INT 21H

Set IOCTL Data

ファンクション 44H
コード 01H

機能 IOCTL データをセットする

コール AH = 44H

AL = 01H

BX ハンドル

DX デバイスデータ (DH=0)

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクション

06H 無効なハンドル

キャリーフラグがセットされない場合

DX デバイスデータ

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

デバイスコントロールデータをセットします。AL は、01H でなければなりません。BX はハンドルでなければなりません。DH は、00H でなければなりません。

デバイスデータの2バイトは、DX の内容にセットされます。デバイスデータのビット7によってハンドルがファイルを表すかデバイスを表すかが決まり、他のビットの意味も異なります。

・デバイス (ビット7=1) の場合

ビット 値 意味

15 予備

14 1 この装置はファンクション 44 H, コード 02 H (IOCTL キャラクタを送る) と 03 H (IOCTL キャラクタを受け取る) を通して、コントロールストリングを処理できます。このビットは読み出すことはできますが書き込むことはできません。

13~8 予備

7 1 ハンドルはデバイスを表す

6 0 EOF を入力する

5 1 コントロールキャラクタをチェックしない

0 コントロールキャラクタをチェックする

4	1	予備
3	1	クロックデバイス
2	1	NUL デバイス
1	1	コンソール出力
0	1	コンソール入力

ビット 5 がチェックできるコントロールキャラクタは, CTRL-C, CTRL-P, CTRL-S, CTRL-Z で, データとして扱うか, コントロールキャラクタとして扱うかを決めます。ビット 5 をセットして CTRL-C をデータとして扱う場合, ファンクション 33 H (<CTRL-C>検査のセット/リセット) または MS-DOS の BREAK コマンドで, CTRL-C をチェックしないようにしなければなりません。

・ファイル (ビット 7=0) の場合

ビット 値 意味

15~8		予備
7	0	ハンドルはファイルを表す
6	0	書き込まれたファイル
5~0		デバイス番号 (00H=A, 01H=B……)

エラーが起きた場合, キャリーフラグがセットされ, AX にエラーコードを返します。

エラーリターン

AX

01H=AL が 01H でないか, AL が 01H で DH が 00H でない,

06H=BX のハンドルがオープンされていないか, 無効である。

マクロ定義

```

ioctl_data      macro      code, handle
                 mov        bx, handle
                 mov        al, code
                 mov        ah, 44H
                 int        21H
                 endm

```

例

ファンクション 44H, コード 00H を参照してください。

INT 21H

Receive IOCTL Character

ファンクション 44H
コード 02H**機能** IOCTL キャラクタを受け取る**コール** AH = 44H

AL = 02H

BX ハンドル

CX コントロールデータのバイト数

DS: DX バッファのポインタ

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクション

06H 無効なハンドル

キャリーフラグがセットされない場合

AX 転送されたバイト数

解説

コントロールデータをキャラクタデバイスから受け取ります。AL は、02H でなければなりません。BX は、プリンタやシリアルポートのようなキャラクタデバイスのハンドルでなければなりません。CX は、読み取るべきコントロールデータのバイト数です。DX は、データバッファのオフセットアドレスです(DS は、セグメントアドレス)。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = AL が 02H でないか、デバイスがファンクションに適合しない。

06H = BX のハンドルがオープンされていないか、実在しない。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

マクロ定義	ioctl_char	macro	code, handle, buffer
		mov	bx, handle
		mov	dx, offset buffer
		mov	al, code
		mov	ah, 44H
		int	21H
		endm	

例

このファンクションはデバイスドライバの IOCTL コントロールデータに依存しますのでプログラムは省略します。

INT 21H

Send IOCTL Character

ファンクション 44H
コード 03H

機 能 IOCTL キャラクタを送る

コール AH = 44H
AL = 03H
BX ハンドル
CX コントロールデータのバイト数
DS:DX バッファのポインタ

リターン キャリーフラグがセットされている場合
AX = 01H 無効なファンクション
06H 無効なハンドル
キャリーフラグがセットされない場合
AX 転送されたバイト数

解 説 IOCTL コントロールデータをキャラクタデバイスに送ります。AL は、03H でなければなりません。BX は、プリンタやシリアルポートのようなキャラクタデバイスのハンドルでなければなりません。CX は、書き込むべきコントロールデータのバイト数です。DX は、データバッファのオフセットアドレスです(DS は、セグメントアドレス)。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H=AL が 03H でないか、デバイスがファンクションに適合しない。
06H=BX のハンドルがオープンされていないか、実在しない。

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

マクロ定義	<code>ioctl_char</code>	<code>macro</code>	<code>code, handle, buffer</code>	
		<code>mov</code>	<code>bx, handle</code>	
		<code>mov</code>	<code>dx, offset buffer</code>	
		<code>mov</code>	<code>al, code</code>	
		<code>mov</code>	<code>ah, 44H</code>	
		<code>int</code>	<code>21H</code>	
		<code>endm</code>		

例 このファンクションはデバイスドライバの IOCTL コントロールデータに依存しますので省略します。

INT 21H

Receive IOCTL Block

ファンクション 44H
コード 04H**機能** IOCTL ブロックを受け取る**コール** AH = 44H

AL = 04H

BL ドライブ番号 (00H = カレント,
01H = A……)

CX コントロールデータのバイト数

DS: DX バッファのポインタ

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクション

05H 無効なドライブ番号

キャリーフラグがセットされない場合

AX 転送されたバイト数

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

コントロールデータをブロックデバイスから受け取ります。AL は、04 H でなければなりません。BL は、ドライブ番号 (00 H = カレント, 01 H = A……) でなければなりません。CX は転送されるべきコントロールデータのバイト数です。DX は、データバッファのオフセットアドレスです (DS は、セグメントアドレス)。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。ファンクション 44 H, コード 00 H 実行の結果、ビット 14 が 1 であれば、そのドライバは IOCTL をサポートしています。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = AL が 04H でないか、デバイスがファンクションに適合しない。

05H = BL のドライブ番号が無効。

マクロ定義	<code>ioctl_block</code>	<code>macro</code>	<code>code, drive, buffer</code>
		<code>mov</code>	<code>bl, drive</code>
		<code>mov</code>	<code>dx, offset buffer</code>
		<code>mov</code>	<code>al, code</code>
		<code>mov</code>	<code>ah, 44H</code>
		<code>int</code>	<code>21H</code>
		<code>endm</code>	

例

このファンクションはデバイスドライバの IOCTL コントロールデータに依存しますのでプログラムを省略します。

INT 21H

Send IOCTL Block

ファンクション 44H
コード 05H

機能 IOCTL ブロックを送る

コール AH = 44H
AL = 05H
BL ドライブ番号 (00H = カレント,
01H = A……)
CX コントロールデータのバイト数
DS:DX バッファのポインタ

リターン キャリーフラグがセットされている場合
AX = 01H 無効なファンクション
05H 無効なドライブ番号
キャリーフラグがセットされない場合
AX 転送されたバイト数

解説 コントロールデータをブロックデバイスに送ります。AL は、05 H でなければなりません。BL は、ドライブ番号 (00 H = カレント, 01 H = A……) でなければなりません。CX は転送すべきコントロールデータのバイト数です。DX は、データバッファのオフセットアドレスです (DS は、セグメントアドレス)。

AX は、転送されたバイト数を返します。デバイスドライバは、IOCTL インターフェイスをサポートしているものでなければなりません。ファンクション 44 H, コード 00 H 実行の結果、ビット 14 が 1 であれば、そのドライバは IOCTL をサポートしています。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = AL が 05H でないか、デバイスがファンクションに適合しない。
05H = BL のドライブ番号が無効。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGS _H	FLAGS _L

CS
DS
SS
ES

マクロ定義

ioctl_block

macro

code, drive, buffer

mov

bl, drive

mov

dx, offset buffer

mov

al, code

mov

ah, 44H

int

21H

endm

例

このファンクションはデバイスドライバの IOCTL コントロールデータに依存しますのでプログラムは省略します。

INT 21H

Get Input IOCTL Status

ファンクション 44H
コード 06H

機 能 入力ステータスのチェック

コール AH = 44H
AL = 06H
BX ハンドル

リターン キャリーフラグがセットされている場合
AX = 01H 無効なファンクション
05H アクセスが否定されました
06H 無効なハンドル
0DH 無効なデータ
キャリーフラグがセットされない場合
AL = 00H レディ状態ではない
FFH レディ

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 ハンドルがレディ状態かどうかをチェックします。AL は、06H でなければなりません。BX は、ハンドルでなければなりません。AL の返す値とステータスの関係はつぎのとおりです。

値	デバイスでの意味	入力ファイルでの意味
00H	レディ状態ではない	ポインタが EOF を指している
FFH	レディ状態	レディ状態

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = AL が 06H でない。
05H = アクセスが否定された。
06H = BX の値が無効か、ハンドルがオープンされている。
0DH = 無効なデータ

マクロ定義	ioctl_status	macro	code, handle
		mov	bx, handle
		mov	al, code
		mov	ah, 44H
		int	21H
		endm	

例 つぎのプログラムは、ハンドルの入力ステータスがレディ状態か、ポインタが EOF を指しているかを表示します。

```

stdin      equ      0
stdout     equ      1
;
message    db        "File is"
ready      db        "ready."
at_eof     db        "at EOF."
crlf       db        0DH, 0AH
;
func_4406H: write_handle stdout, message, 8 ; message を表示
            jc          write_error
            ioctl_status 6, stdin           ; 入力ステータスをチェック
            jc          ioctl_error
            cmp         al, 0               ; 入力ステータスはレディか?
            jne         not_eof            ; はいの時, not_eof へ
            write_handle stdout, at_eof, 7 ; at_eof を表示(40H)
            jc          write_error
            jmp         all_done            ; all_done へ
not_eof:    write_handle stdout, ready, 6   ; ready を表示(40H)
all_done:   write_handle stdout, crlf, 2    ; crlf を表示(40H)
            jc          write_error

```

INT 21H

Get Output IOCTL Status

ファンクション 44H
コード 07H

機能 出力ステータスのチェック

コール AH = 44H
AL = 07H
BX ハンドル

リターン キャリーフラグがセットされている場合
AX = 01H 無効なファンクション
05H アクセスの否定
06H 無効なハンドル
0DH 無効なデータ
キャリーフラグがセットされていない場合
AL = 00H レディ状態ではない
FFH レディ

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSh	FLAGSl

CS
DS
SS
ES

解説 ハンドルがレディ状態かどうかをチェックします。AL は、07H でなければなりません。BX は、ハンドルでなければなりません。AL の返す値とステータスの関係はつぎのとおりです。

値	デバイスでの意味	出力ファイルでの意味
00H	レディ状態ではない	レディ状態
FFH	レディ状態	レディ状態

出力ファイルは、たとえディスクがいっぱいでも、レディ状態を返します。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

- 01H = AL が 07H でない。
- 05H = アクセスが否定された。
- 06H = BX の値が無効か、ハンドルがオープンされている。
- 0DH = 無効なデータ。

マクロ定義	<code>ioctl_status</code>	<code>macro</code>	<code>code, handle</code>
		<code>mov</code>	<code>bx, handle</code>
		<code>mov</code>	<code>al, code</code>
		<code>mov</code>	<code>ah, 44H</code>
		<code>int</code>	<code>21H</code>
		<code>endm</code>	

例 ファンクション 44H, コード 06H を参照してください。

INT 21H

IOCTL Is Changeable

ファンクション 44H
コード 08H**機能** IOCTL の交換性**コール** AH = 44H

AL = 08H

BL ドライブ番号 (00H = カレント,
01H = A……)**リターン** キャリーフラグがセットされている場合

AX = 01H 無効なファンクション

0FH 無効なドライブ番号

キャリーフラグがセットされない場合

AX = 00H 交換可能

01H 交換不可能

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 BL は、ドライブ番号 (00 H = カレント, 01 H = A……) でなければなりません。AX が 01 H の場合、ハードディスクのように交換できないドライブです。AX が 00 H の場合、通常のディスクのように交換できるドライブです。

このファンクションが実行されるときにディスクを交換するか否かのメッセージが出ます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = デバイスがこのコールでサポートされていない。

0FH = BL で指定されたドライブ番号が無効。

マクロ定義	ioctl_change	macro	drive
		mov	bl, drive
		mov	al, 08H
		mov	ah, 44H
		int	21H
		endm	

例

つぎのプログラムは、カレントディスクが交換できるかどうかを調べ、交換できないディスクの場合は作業を続け、交換できる場合はディスクを差し換える旨のメッセージを出します。

```

stdout      equ      1
;
message     db        "Please replace disk in drive"
drives      db        "ABCD"
crlf        db        0DH, 0AH
;
func_4408H:  ioctl_change 0          ; IOCTL の交換性のチェック
              jc          ioctl_error
              cmp         ax, 0      ; カレントドライブの交換は可能か?
              jne         continue  ; いいえの時、処理を継続
              write_handle stdout, message, 29 ; はいの時、message を表示(40H)
              jc          write_error
              current_disk          ; カレントディスクを得る(19H)
              xor         bx, bx     ; インデックスをクリア
              mov         bl, al      ; カレントドライブ番号をセット
              display_char drives[bx] ; カレントドライブをスクリーン
                                      ; に出力(02H)
              write_handle stdout, crlf, 2 ; crlf を表示(40H)
              jc          write_error
continue:
;          (Further processing here)

```


INT 21H

IOCTL Is Redirected Block

ファンクション 44H
コード 09H**機能** IOCTL リダイレクトブロック**コール** AH = 44H

AL = 09H

BL ドライブ番号 (00 H = カレント,
01 H = A……)**リターン** キャリーフラグがセットされている場合

AX = 01H 無効なファンクション

0FH 無効なドライブ番号

キャリーフラグがセットされない場合

DX デバイスアトリビュートワード

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

このファンクションは、ドライブ名が MS-Networks のワークステーション (ローカル) のドライブであるか、サーバ (リモート) へリダイレクトされているかをチェックします。BL は、ドライブ番号 (00 H = カレント, 01 H = A……) でなければなりません。

ブロックデバイスがローカルの場合、DX はデバイスヘッダのアトリビュートワード (2 バイト) を返します。ブロックデバイスがリモートの場合、ビット 12 だけがセットされ (1000H)、他のビットは 0 (予備) です。

アプリケーションプログラムでは、ビット 12 をチェックすることはできません。したがってローカル、リモート、デバイスの区別ができません。

エラーが起きた場合、キャリーフラグがセットされ、AX はエラーコードを返します。

エラーリターン

AX

01H = このシステムコールを使うためには、ファイルシェアリング (SHARE.EXE) がロード (常駐) されていなければなりません。

0FH = BL で指定されたドライブ番号が無効。

マクロ定義

```

ioctl_rblock    macro    drive
                 mov      bl, drive
                 mov      al, 09H
                 mov      ah, 44H
                 int       21H
                 endm

```

例

つぎのプログラムはドライブ B がローカルか、リモートかをチェックし、適切なメッセージを表示します。

```

stdout          equ      1
;
message         db        "Drive B: is"
loc             db        "local."
rem            db        "remote."
crlf           db        0DH, 0AH
;
func_4409H: write_handle stdout, message, 12 ; message を表示
             jc          write_error
             ioctl_rblock 2                  ; ドライブ B がローカルかリモート
                                                ; かをチェック
             jc          ioctl_error
             test        dx, 1000h           ; ビット 12 がセットされているか?
             jnz         not_loc             ; はいの時、リモートで、not_loc へ
             write_handle stdout, loc, 6     ; loc を表示(40H)
             jc          write_error
             jmp         done
not_loc:        write_handle stdout, rem, 7   ; rem を表示(40H)
             jc          write_error
done:          write_handle stdout, crlf, 2   ; crlf を表示(40H)
             jc          write_error

```

INT 21H

IOCTL Is Redirected Handle

ファンクション 44H
コード 0AH**機能** IOCTL リダイレクトハンドル**コール** AH = 44H
AL = 0AH
BX ハンドル**リターン** キャリーフラグがセットされている場合
AX = 01H 無効なファンクションコード
06H 無効なハンドル
キャリーフラグがセットされていない場合
DX IOCTL ビットフィールド

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 このファンクションは、ファイルが MS-Networks のワークステーション（ローカル）のファイルまたはデバイスであるか、サーバへリダイレクトされているかをチェックします。BX は、ファイルハンドルでなければなりません。DX は IOCTL ビットフィールドを返します。ビット 15 が 1 の場合、ハンドルはリモートファイルかデバイスです。

アプリケーションプログラムでは、ビット 15 をチェックすることはできません。したがってローカル、リモート、デバイスの区別ができません。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = このファンクションリクエストを実行するには、MS-Networks が稼動していなければなりません。

06H = BX で指定されたドライブ番号が無効。

マクロ定義	ioctl_rhandle	macro	handle
		mov	bx, handle
		mov	al, 0AH
		mov	ah, 44H
		int	21H
		endm	

例

つぎのプログラムはハンドル5がローカルかリモートのファイルか、デバイス
かを表示します。

```

stdout      equ      1
;
message      db      "Handle 5 is"
loc          db      "local."
rem          db      "remote."
crlf         db      0DH, 0AH
;
func_440AH:  write_handle stdout, message, 12 ; message を表示
              jc          write_error
              ioctl_rhandle 5                  ; ハンドル5がローカルかリモート
                                              ; かをチェック

              jc          ioctl_error
              test        dx, 8000h             ; ビット5がセットされているか?
              jnz         not_loc               ; はいの時、リモートである、not_locへ
              write_handle stdout, loc, 6       ; locを表示(40H)
              jc          write_error
              jmp         done
not_loc:     write_handle stdout, rem, 7        ; remを表示(40H)
              jc          write_error
done:        write_handle stdout, crlf, 2       ; crlfを表示(40H)
              jc          write_error

```

INT 21H

IOCTL Retry

ファンクション 44H
コード 0BH

機能 IOCTL リトライ

コール AH = 44H

AL = 0BH

DX リトライの回数

CX 待ち時間

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクションコード

キャリーフラグがセットされない場合

エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

このファンクションは、ファイルの共有違反が発生したときに、MS-DOS が行うリトライの回数をセットします。DX は、リトライの回数でなければなりません。CX は、リトライの間隔の時間でなければなりません。

MS-DOS は、このファンクションによって変更されない限り、リトライを3回行います。指定されたリトライをセットした後、要求されたプロセスのために、MS-DOS は割り込みタイプ 24H を実行します。

CX で与えた待ち時間に対して、実際の時間は機種によって異なります。それは、MS-DOS が用意した待ち時間のループ、CPU の処理速度とクロックサイクルに依存します。ユーザーが実際の時間を知り、それをもとに設定したい場合は、リトライの回数を1にして、待ち時間をいろいろ変えるのも1つの手段です。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H=このシステムコールを使うためには、ファイルシェアリング (SHARE.EXE) がロード (常駐) されていなければなりません。

マクロ定義	ioctl_retry	macro	retries, wait
		mov	dx, retries
		mov	cx, wait
		mov	al, 0BH
		mov	ah, 44H
		int	21H
		endm	

例 つぎのプログラムはリトライの回数を 10 にし、待ち時間を 1000 にします。

```
func_440BH: ioctl_retry    10,1000    ; ディスクアクセスのリトライ
                                   ; 回数を 10 にセット
                                   jc      error
```


INT 21H

Generic IOCTL (for handles)

ファンクション 44H
コード 0CH**機能** 一般 IOCTL (ハンドル用)

コール AH = 44H
 AL = 0CH
 BX ハンドル
 CH = 05H カテゴリコード(プリンタデバイス)
 CL ファンクション (マイナー) コード
 DS:DX データバッファへのポインタ

リターン キャリーフラグがセットされている場合
 AX = 1 無効なファンクションコード
 キャリーフラグがセットされない場合
 エラーなし

解説 このシステムコールは、“PRINT TIL BUSY”がサポートされているプリンタドライバに対して、プリンタへの出力の繰り返し回数を設定または取得します。

CL=45H ならば、このコールは、プリンタに対する繰り返し回数をセットします。CL=65H ならば、このコールは、プリンタに対する繰り返し回数を取得します。

DS:DX は “PRINT TIL BUSY” ループの繰り返し回数が格納されているワードをポイントします。これは、デバイスドライバがデバイスから “READY” シグナルが返されるまでデバイス BUSY を待つ回数です。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

INT 21H

Generic IOCTL (for block devices)

ファンクション 44H
コード 0DH

機能 一般 IOCTL (ブロックデバイス用)

コール AH = 44H

AL = 0DH

BL デバイス番号(0= カレント, 1=A, ..., など)

CH = 08H カテゴリ (メジャー) コード

CL ファンクション (マイナー) コード

DS:DX パラメータブロック-1 へのポインタ

リターン キャリーフラグがセットされた場合

AX = 1 無効なファンクションコード

2 無効なドライブ

キャリーフラグがセットされない場合

エラーなし

解説 ファンクションコードは、次のうちの一つです。

コード 解説

40 H デバイスパラメータのセット

60 H デバイスパラメータの取得

41 H 論理デバイス上のトラックのライト (書き出し)

61 H 論理デバイス上のトラックのリード (読み込み)

42 H 論理デバイス上のトラックのフォーマット

62 H 論理デバイス上のトラックのベリファイ

注意：論理デバイスのリード、ライト、フォーマット、ベリファイの前に、デバイスパラメータのセットをしなければなりません。

論理デバイスのリード、ライト、フォーマットまたはベリファイを行いたい場合は、次の手順で行ってください。

- デバイスパラメータの取得を使用して、ドライブパラメータをセーブします。
- デバイスパラメータのセットを使用して、希望するドライブパラメータをセットします。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

- I/Oオペレーションを実行します。
- デバイスパラメータのセットを使用して、オリジナルなドライブパラメータを復元します。

■ デバイスパラメータのセット (ファンクション 440DH, CL=40H)

CL=40H のとき、パラメータブロックは、次のようなフィールドフォーマットです。

バイト	特殊ファンクション
バイト	デバイスタイプ
ワード	デバイス属性
ワード	シリンダ数
バイト	メディアタイプ
デバイス BPB	
トラックレイアウト	

これらのフィールドは、次のような意味を持ちます。

特殊ファンクションフィールド：

各ビットごとの、値と意味は次のとおりです。

ビット	値	意味
0	0	デバイス BPB フィールドには、このデバイスに対する新しいデフォルトの BPB を含んでいる。もし、デバイスのセットのコールが以前にこのビットをセットしたならば、Build BPB は実際のメディア BPB を返し、さもなければ、デバイスに対するデフォルト BPB を返す。
	1	すべての BUILD BPB リクエストの結果として、デバイス BPB が返される。
1	0	パラメータブロックのすべてのフィールドのリード。
	1	トラックレイアウトフィールドを除く、すべてのフィールドのパラメータが無視される。
2	0	トラック上のセクタサイズが同じでない (この設定は使用すべきではない)。
	1	トラック上のセクタサイズはすべて同じであり、セクタ番号の範囲は、1 から現在のトラック上の総数までである。このビ

ットは、常にセットするべきである。

3, 4, 5, 6, 7 0 これらのビットは、0 でなければならない。

デバイスタイプフィールド：

このバイトは、物理デバイスを記述し、デバイスによってセットされます。その値と意味は、次のとおりです。

値	意味
0	320 / 360K バイト
1	—
2	640K / 720K バイト
3	256K バイト (8 インチ単密度)
4	1 メガバイト
5	固定ディスク
6	—
7	その他

デバイス属性フィールド：

各ビットごとの、値と意味は次のとおりです。

ビット	値	意味
0	0	メディアは、交換可能。
	1	メディアは、交換不可能。
1	0	ディスクチェンジラインは、サポートされていない。 (ドアロックがサポートされていない)
	1	ディスクチェンジラインは、サポートされている。 (ドアロックがサポートされている)
2～15	0	これらのビットは 0 でなければならない。

シリンダ数フィールド：

このフィールドは、物理デバイスがサポートできるシリンダ数の最大値を示します。この情報は、デバイスによってセットされます。

メディアタイプフィールド：

複数の種類のメディアを使用できるドライブのために、このフィールドは (デバイスに依存) どの種類のメディアがドライブにセットされているかを示します。

デバイス BPB フィールド：

特殊ファンクションフィールドのビット 0 がクリアされた場合、このフィールドの BPB はデバイスの新しいデフォルトの BPB です。

特殊ファンクションフィールドのビット 0 がセットされた場合、デバイスドライバは、BUILD BPB リクエストの後でこのフィールドに BPB を返します。

トラックレイアウトフィールド：

このフィールドは、各論理デバイスの可変長テーブルと、期待されるメディアトラック上のセクタのレイアウトを示します。このフィールドのフォーマットは、次のとおりです。

ワード	セクタカウント	——	セクタの総数
ワード	セクタ番号	——	セクタ 1
ワード	セクタサイズ	——	セクタ 1
ワード	セクタ番号	——	セクタ 2
ワード	セクタサイズ	——	セクタ 2

⋮

ワード	セクタ番号	——	セクタ n
ワード	セクタサイズ	——	セクタ n

セクタカウントフィールドは、セクタの総数を示します。各セクタ番号は一意で、1 からセクタ総数 (n) でなければなりません。

特殊ファンクションフィールドのビット 2 がセットされた場合は、すべてのセクタサイズは同じでなければなりません。

■デバイスパラメータの取得 (ファンクション 440DH, CL=60H)

CL=60H のとき、パラメータブロックフィールドは CL=40H のように、同じフィールドレイアウトです。しかし、いくつかのフィールドは、異なった意味を持っています。それらは、つぎのような意味を持っています。

特殊ファンクションフィールド：

各ビットごとの、値と意味は次のとおりです。

ビット	値	意味
0	0	デバイスに対するデフォルトの BPB を返す
	1	BUILD BPB リクエストが返した BPB を返す
1, 2, 3, 4, 5, 6, 7	0	これらのビットは 0 でなければならない

トラックレイアウトフィールド：

デバイスパラメータの取得コールは、このフィールドを使用しません。

■論理デバイス上のトラックのリード／ライト

(ファンクション 440DH, CL=61H/CL=41H)

論理デバイス上のトラックヘライト（書き出し）するには、CL=41H をセットします。論理デバイス上のトラックをリード（読み込み）するには、CL=61H をセットします。

CL=41H または CL=61H のとき、パラメータブロックのフォーマットは次のとおりです。

バイト	特殊ファンクション
ワード	ヘッド
ワード	シリンダ
ワード	第1セクタ
ワード	セクタ数
2ワード	転送アドレス

これらのフィールドの内容は、次のとおりです。

特殊ファンクションフィールド：

このバイトは 0 です。

ヘッドフィールド：

このフィールドは、書き込み、または読み出しを行うときのヘッド番号を含みます。

シリンダフィールド：

このフィールドは、書き込み、または読み出しを行うときのシリンダ番号を含みます。

ファーストセクタフィールド：

このフィールドは、書き込み、または読み出しを行うときの最初のセクタ番号を持ちます。このセクタは0から数え始められるため、4番目のセクタは3と数えられるわけです。

セクタ番号フィールド：

このフィールドは、セクタの総数が含まれます。

転送アドレスフィールド：

このフィールドには、格納されている書き出すべきデータまたは現在読み込まれているデータの、アドレスが含まれています。

■論理デバイス上のトラックのフォーマット／ベリファイ

(ファンクション 440DH, CL=42H/CL=62H)

論理デバイス上のトラックの、フォーマットとベリファイをする場合は、CL=42H をセットします。論理デバイス上のトラックをベリファイする場合は、CL=62H をセットします。

CL=42H または CL=62H のとき、パラメータブロックのフォーマットは、次のとおりです。

バイト	特殊ファンクション
ワード	ヘッド
ワード	シリンダ

これらのフィールドの意味は、次のとおりです。

特殊ファンクションフィールド：

このバイトは、0 でなければなりません。

ヘッドフィールド：

このフィールドは、フォーマットまたはベリファイを実行する、ヘッド番号を含みます。

シリンダフィールド：

このフィールドは、フォーマットまたはベリファイを実行する、シリンダ番号を含みます。

INT 21H

Get/Set Logical Drive Map

ファンクション 44H
コード 0EH, 0FH

- 機能** 論理ドライブマップの取得/設定
- コール** AH = 44H
AL = 0EH 論理ドライブマップの取得
0FH 論理ドライブマップの設定
BX ドライブ番号 (0=カレント, 1=A, ..., など)
- リターン** キャリーフラグがセットされた場合
AX = 01H 無効なファンクションコード
0FH 無効なドライブ番号
キャリーフラグがセットされない場合
AL = 論理デバイスは物理的にマップされた (=0, 1ドライブがこの物理ドライブに割り当てられた)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

論理ドライブマップの取得は、DOSに対して、現在どの論理ドライブが物理デバイスにマップされているかを質問します。論理ドライブマップの設定は、現在物理デバイスにマップされているドライブを変更します。これらのファンクションは、ディスクドライブが1台のシステムでのみ有効です。

アプリケーションでは、これらのファンクションによって、DOSが現在示すドライブ中の正しいフロッピーディスクの場所を無効にして、他の論理ドライブにアクセスすることができます。

論理ドライブが現在どの物理デバイスにマップされているかどうかを検出するためには、プログラムは、ファンクション 440EH または 440FH (論理ドライブマップの取得/設定) のコールの後で、AL の値を調べる必要があります。

INT 21H

Duplicate File Handle

ファンクション 45H

機能 ファイルハンドルの二重化

コール AH = 45H

BX ファイルハンドル

リターン キャリーがセットされた場合

AX = 04H オープンされているファイルが多すぎる。

06H 無効なハンドル

キャリーがセットされない場合

AX 新規のファイルハンドル

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

1つのファイルに追加するハンドルを作成します。BX は、オープンされたファイルのハンドルでなければなりません。

MS-DOS は新しいハンドルを AX に返します。すでにオープンされている BX で指定したファイルハンドルを取り出し、同じファイルを示す新規のファイルハンドルを返します（2つのファイルのリード/ライトポインタは同じところを指します）。

このファンクション実行後、どちらか1方のリード/ライトポインタを移動すると、もう1方のポインタもまた移動します。このファンクションは通常、標準入力（ハンドル0）と標準出力（ハンドル1）をリダイレクトとして扱います。

エラーが起きた場合、キャリーフラグがセットされ、AX にはエラーコードが返されます。

エラーリターン

AX

04H = 現在のプロセスで使用可能なハンドルが存在しなかったか、また内部システムテーブルに空き領域がありませんでした。

06H = BX に渡されたハンドルは、現在オープンされていません。

マクロ定義	xdup	macro	handle
		mov	bx, handle
		mov	ah, 45H
		int	21H
		endm	

例 つぎのプログラムは、標準出力（ハンドル1）を“DIRFILE”というファイルに定義しなおし、ディレクトリを出力するための子プロセスを起動し、標準入力をハンドル1に戻します。

pgm_file	db	"command. com", 0	
cmd_line	db	9, "/c dir/w", 0DH	
parm_blk	db	14 dup (0)	
path	db	"dirfile", 0	
dir_file	dw	?	; ハンドル用
sav_stdout	dw	?	; ハンドル用
;			
func_45H	set_block	last_inst	; 割り当てられたブロックの変更(4AH)
	jc	error_setblk	
	create_handle	path, 0	; ハンドルの作成(3CH)
	jc	error_create	
	mov	dir_file, ax	; ハンドルをセーブ
	xdup	1	; ファイルハンドルを二重化
	jc	error_xdup	
	mov	sav_stdout, ax	; ハンドルをセーブ
	xdup2	dir_file, 1	; ハンドルを強制的に二重化(46H)
	jc	error_xdup2	
	exec	pgm_file, cmd_line, parm_blk	; 子プロセスを起動(4BH)
	jc	error_exec	
	xdup2	sav_stdout, 1	; ハンドルを強制的に二重化(46H)
	jc	error_xdup2	
	close_handle	sav_stdout	; ハンドルのクローズ(3EH)
	jc	error_close	
	close_handle	dir_file	; ハンドルのクローズ(3EH)
	jc	error_close	

INT 21H

Force Duplicate File Handle

ファンクション 46H

機能 ファイルハンドルの強制二重化

コール AH = 46H

BX 既存のファイルハンドル

CX 新規のファイルハンドル

リターン キャリーがセットされた場合

AX = 04H オープンされているファイル
が多すぎる。

06H 無効な処理

キャリーがセットされない場合

エラーなし

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

オープンしたファイルと既に連結(二重化)されている他のハンドルを、指定されたハンドルと強制的に二重化させます。BX は、オープンされたファイルのハンドルでなければなりません。CX は、新規のハンドルでなければなりません。

すでにオープンされているファイルハンドルを取り出し、同じ位置の同じファイルを示す新規のファイルハンドルを返します。CX のファイルハンドルが既にオープンされている場合、まず、そのハンドルがクローズされます。

このファンクション実行後、どちらか一方のリード/ライトポインタを移動するともう一方のポインタもまた移動します。このファンクションは通常、標準入力(ハンドル0)と標準出力(ハンドル1)をリダイレクトとして扱います。

エラーが起きた場合、キャリーフラグがセットされ、AX にはエラーコードが返されます。

エラーリターン

AX

04H=現在のプロセスで使用可能な解放されたハンドルが存在しないか、
または内部システムテーブルに空き領域がありませんでした。

06H=BX に渡されたハンドルは、現在オープンされていません。

マクロ定義

xdup2

```
macro      handle1, handle2
mov        bx, handle1
mov        cx, handle2
mov        ah, 46H
int        21H
endm
```

例

つぎのプログラムは、標準出力（ハンドル1）を“DIRFILE”というファイルに定義しなおし、ディレクトリを出力するための子プロセスを起動し、標準入力をハンドル1に戻します。

```
pgm_file   db      "command. com", 0
cmd_line   db      9, "/c dir/w", 0DH
parm_blk   db      14 dup (0)
path       db      "dirfile", 0
dir_file   dw      ?           ; ハンドル用
sav_stdout dw      ?           ; ハンドル用
;
func_46H:  set_block  last_inst      ; 割り当てられたメモリブロックの変更(4AH)
           jc        error_setblk
           create_handle path, 0      ; ハンドルの作成(3CH)
           jc        error_create
           mov       dir_file, ax     ; ハンドルをセーブ
           xdup      1                ; ファイルハンドルを二重化(45H)
           jc        error_xdup
           mov       sav_stdout, ax   ; ハンドルをセーブ
           xdup2     dir_file, 1      ; ハンドルを強制的に二重化
           jc        error_xdup2
           exec      pgm_file, cmd_line, parm_blk ; 子プロセスを起動(48H)
           jc        error_exec
           xdup2     sav_stdout, 1    ; ハンドルを強制的に二重化
           jc        error_xdup2
           close_handle sav_stdout    ; ハンドルのクローズ(3EH)
           jc        error_close
           close_handle dir_file      ; ハンドルのクローズ(3EH)
           jc        error_close
```


INT 21H

Get Current Directory

ファンクション 47H

- 機能** カレントディレクトリを得る
- コール** AH = 47H
DS:SI 64 バイトのメモリ領域に対する
ポインタ
DL ドライブ番号
- リターン** キャリーがセットされた場合
AX = 0FH 無効なドライブ
キャリーがセットされない場合
エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

- 解説** 指定したドライブのカレントディレクトリのパス名を返します。DL は、ドライブ番号 (00H = カレント, 01H = A ...) でなければなりません。SI は、64 バイトのメモリ領域のオフセットアドレス (DS は、セグメントアドレス) です。

DS:SI で指定するメモリ領域は、ルートディレクトリからの相対位置で表したパス名 (DL で指定したドライブのカレントディレクトリ) の文字列を ASCIZ 文字列としたものです。この文字列は、 \backslash マーク (ルートディレクトリを表す) から始まらず、ドライブ指定も含んでおりません。

エラーが起きた場合キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

0FH = DL で指定されたドライブ番号は無効。

マクロ定義

```

get_dir      macro      drive, buffer
              mov       dl, drive
              mov       si, offset buffer
              mov       ah, 47H
              int       21H
              endm

```

例

つぎのプログラムは、ドライブB上のディスクのカレントディレクトリを表示します。

```

disk         db         "b: $"
buffer       db         64 dup (?)
;
func_47H:    get_dir     2, buffer           ; カレントディレクトリを得る
              jc         error_dir
              display     disk              ; disk をスクリーンに出力(09H)
              display_asciz buffer         ; 章末参照

```

INT 21H

Allocate Memory

ファンクション 48H

機能 メモリの割り当て

コール AH = 48H

BX 割り当てるべきメモリの大きさ (パラグラフ)

リターン キャリーがセットされた場合

AX = 07H メモリ中のデータの破壊

08H 十分な大きさのメモリがない

BX 割り当て可能な最大のメモリサイズ

キャリーがセットされない場合

AX 割り当てられたメモリのセグメント
アドレス (パラグラフ)

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

カレントプロセスに指定された大きさのメモリを割り当てます。BX は割り当てられるべきメモリの大きさ (パラグラフ単位: 1 パラグラフ=16 バイト) でなければなりません。

要求を満たすメモリがある場合、AX に割り当てられたメモリのセグメントアドレスを返します。要求されたメモリがない場合には、BX に割り当て可能な最大のメモリサイズ (パラグラフ単位) を返します。

エラーが起きた場合キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

07H = メモリの内部に、矛盾したデータが含まれています。これはユーザープログラムが所有していないメモリを変更したために発生します。

08H = 使用可能な最大の空きのブロックが要求されたものより小さいか、または空きのブロックが存在しません。

マクロ定義	allocate-memory	macro	bytes
		mov	bx, bytes
		mov	cl, 4
		shr	bx, cl
		inc	bx
		mov	ah, 48H
		int	21H
		endm	

例

つぎのプログラムは、“TEXTFILE.ASC”というファイルをオープンし、ファンクション 42H（ファイルポインタの移動）で、サイズを求めます。つぎに、そのファイルサイズでメモリブロックを割り当て、割り当てたメモリにファイルを読み込みます。最後に、割り当てたメモリを解放します。

path	db	"textfile. asc", 0	
msg1	db	"File loaded into allocated memory block.",	
		0DH, 0AH	
msg2	db	"Allocated memory now being freed	
		(deallocated).", 0DH, 0AH	
handle	dw	?	
mem_seg	dw	?	
file_len	dw	?	
;			
func_48H:	open_handle	path, 0	; ハンドルのオープン(3DH)
	jc	error_open	
	mov	handle, ax	; ハンドルをセーブ
	move_ptr	handle, 0, 0, 2	; ファイルポインタを移動(42H)
	jc	error_move	
	mov	file_len, ax	; ファイルサイズをセーブ
	set_block	last_inst	; 割り当てられたメモリブロック
			; の変更(4AH)
	jc	error_setblk	
	allocate_memory	file_len	; メモリを割り当てる
	jc	error_alloc	
	mov	mem_seg, ax	; 新規のメモリのアドレスをセーブ

```
move_ptr    handle, 0, 0, 0      ; ファイルポインタを移動(42H)
jc          error_move
push        ds                  ; DS をセーブ
mov         ax, mem_seg         ; 新規のメモリのセグメントアドレスを得る
mov         ds, ax              ; 新規メモリに DS をセット
read_handle cs:handle, 0, cs:file_len ; 新規に割り当てられたメモリにファイルを読み込む
pop         ds                  ; DS をリストア
jc          error_read
; (CODE TO PROCESS FILE GOES HERE)
write_handle stdout, msg1, 42    ; msg1 を表示(40H)
jc          write_error
free_memory mem_seg             ; 割り当てられたメモリを解放(49H)
jc          error_freemem
write_handle stout, msg2, 49     ; msg2 を表示(40H)
jc          write_error
```

INT 21H

Free Allocated Memory

ファンクション 49H

機能	割り当てられたメモリの解放
コール	AH = 49H ES 解放すべきメモリ領域のセグメント アドレス
リターン	キャリーがセットされた場合 AX = 07H メモリ中のデータの破壊 09H 無効なブロック キャリーがセットされない場合 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解説 ファンクション 48H（メモリの割り当て）で、先に割り当てられたメモリブロックを解放（利用可能に）します。ES は、解放されるメモリブロックのセグメントアドレスでなければなりません。

エラーが起きた場合キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

07H = メモリの内部に、矛盾したデータが含まれています。これはユーザープログラムが所有していないメモリを変更したために発生します。

09H = ES 内に渡されたブロックは、メモリアロケーションによって割り当てられたものではありません。

マクロ定義	free-memory	macro	seg-addr
		mov	ax, seg-addr
		mov	es, ax
		mov	ah, 49H
		int	21H
		endm	

例

つぎのプログラムは、“TEXTFILE.ASC”というファイルをオープンし、ファンクション 42H（ファイルポインタの移動）で、サイズを求めます。つぎに、そのファイルサイズでメモリブロックを割り当て、割り当てたメモリにファイルを読み込みます。最後に、割り当てたメモリを解放します。

```

path          db          "textfile.asc", 0
msg1          db          "File loaded into allocated memory block.",
                        0DH, 0AH
msg2          db          "Allocated memory now being freed
                        (deallocated).", 0DH, 0AH
handle        dw          ?
mem_seg       dw          ?
file_len      dw          ?
;
func_49H:     open_handle  path, 0          ; ハンドルのオープン(3DH)
              jc          error_open
              mov         handle, ax        ; ハンドルをセーブ
              move_ptr    handle, 0, 0, 2   ; ファイルポインタを移動(42H)
              jc          error_move
              mov         file_len, ax      ; ファイルサイズをセーブ
              set_block   last_inst        ; 割り当てられたメモリブロックの変更
              jc          error_setblk
              allocate_memory file_len      ; メモリの割り当て(48H)
              jc          error_alloc
              mov         mem_seg, ax       ; 新規メモリのアドレスをセーブ
              move_ptr    handle, 0, 0, 0   ; ファイルポインタを移動(42H)
              jc          error_move
              push        ds               ; DS をセーブ
              mov         ax, mem_seg      ; 新規メモリのセグメントアドレスを得る
              mov         ds, ax           ; 新規メモリを DS でポイントする
              read_handle  handle, code, file_len ; 新規に割り当てられたメモリに
                                              ; ファイルを読み込む
              pop         ds               ; DS をリストア
              jc          error_read
              ; (CODE TO PROCESS FILE GOES HERE)

```

```
write_handle stdout, msg1, 42 ; msg1 を表示(40H)
jc          write_error
free_memory mem_seg          ; 割り当てられたメモリを解放
jc          error_freemem
write_handle stdout, msg2, 49 ; msg2 を表示(40H)
jc          write_error
```

INT 21H

Set Block

ファンクション 4AH

機能 割り当てられたメモリブロックの変更

コール AH = 4AH
 ES メモリ領域のセグメントアドレス
 BX 変更したいメモリの大きさ
 (パラグラフ)

リターン キャリーがセットされた場合
 AX = 07H メモリ中のデータの破壊
 08H 十分な大きさのメモリがない
 09H 無効なブロック
 BX 使用可能な最大の大きさ
 キャリーがセットされない場合
 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 割り当てられたメモリブロックの大きさを変更します。ES は、パラグラフ(1パラグラフ=16 バイト)単位のメモリブロックのセグメントアドレスでなければなりません。

MS-DOS は、メモリブロックのサイズを変更しようとします。このファンクションがメモリを拡大させるのに失敗した場合、BX は、使用可能な最大のブロック(パラグラフ単位)を返します。

MS-DOS は、利用可能なメモリのすべてを COM 形式のファイルに割り当ててしまいますので、このコールは、しばしば割り当てられたプログラムのメモリブロックの初期値の縮小に使われます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

07H = メモリの内部に、矛盾したデータが含まれています。これは、ユーザープログラムが所有していないメモリを変更したために発生します。

08H = 指定されたブロックの後に、拡大要求を満たすのに十分な空きメモリがありませんでした。

09H = ES のアドレスが不正です (ES に渡されたブロックは、メモリアロケーションによって割り当てられたものではありません)。

マクロ定義

このマクロは、COM 形式のプログラムの割り当てられたメモリブロックの初期値を縮小 (整理) するものです。プログラムの最後の命令に続く最初のバイトのオフセットをパラメータ (LAST INST はサンプルプログラムを参照してください) として渡し、そのパラメータをパラグラフ単位に換算します。つぎにその計算結果に 17 (1 は、ラウンドアップ用、16 は、256 バイトのスタック用) を加え、SP と BP をそのスタックのポインタにセットします。

```
set_block      macro      last_byte
                mov       bx, offset last_byte
                mov       cl, 4
                shr       bx, cl
                add       bx, 17
                mov       ah, 4AH
                int       21H
                mov       ax, bx
                shl       ax, cl
                dec       ax
                dec       ax
                mov       sp, ax
                endm
```

例

つぎのプログラムは、子プロセスを起動し、DIR コマンドを実行します。

```
pgm_file      db          "command.com", 0
cmd_line      db          9, "/c dir /w", 0DH
parm_blk      db          14 dup (?)
reg_save      db          10 dup (?)
;
func_4AH:     set_block   last_inst      ; 割り当てられたメモリブロックの変更
               exec       pgm_file, cmd_line, parm_blk, 0
               ; 子プロセスを起動し、DIR コマンドを実行(4BH)
```

INT 21H

Load and Execute Program

ファンクション 4BH
コード 00H**機能** プログラムのロードと実行**コール** AH = 4BH

AL = 00H

DS : DX パス名の位置

ES : BX パラメータブロックの位置

リターン キャリーがセットされた場合

AX = 01H 無効なファンクション

02H ファイルが存在しない

04H オープンされているファイルが多すぎる

05H アクセスの否定

08H 十分な大きさのメモリがない

0AH 不正な環境

0BH 不正なフォーマット

キャリーがセットされない場合

エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

DX は、実行可能なプログラムのドライブ名とパス名を表す ASCIZ 文字列のオフセットアドレス (セグメントアドレスは DS) でなければなりません。BX はロードのためのパラメータブロックのオフセットアドレス (セグメントアドレスは ES) でなければなりません。AL は 00H でなければなりません。

MS-DOS がプログラムをロードするのに十分な空きメモリ領域がなければなりません。すべての空きメモリ領域はロードされたときにプログラムに割り当てられるので、ファンクション 4BH、コード 00H を使って他のプログラムをロードし実行する前に、ユーザーはファンクション 4AH (割り当てられたメモリブロックの変更) を使って、メモリ空間を解放しなければなりません。メモリ空間が他の目的で使用されない限り、このファンクションリクエストが実行される前にカレントプロセスによって縮小しなければなりません。

MS-DOS はプログラムをロードするためにプログラムセグメントプレフィクスを作成し、ファンクション 4BH が呼ばれた直後に、終了アドレス、〈CTRL-

C) の抜け出しアドレスをセットします。

つぎに、パラメータブロックのアドレスの内容を示します。

オフセット	バイト長	意味
00H	2	渡される環境のセグメントアドレス。00H の場合、親環境のコピーであることを示します。
02H	4	プログラムセグメントプレフィックスのオフセット 80H のコマンドラインのセグメントアドレス (先の 2 バイト) とオフセットアドレス (続く 2 バイト)。これは、128 バイトを超えない正しいコマンドラインでなければなりません。
06H	4	新しいプログラムセグメントプレフィックス (プログラムセグメントプレフィックスの詳細については第 4 章を参照してください) のオフセット 5CH にある FCB のセグメントアドレス (先の 2 バイト) とオフセットアドレス (続く 2 バイト)。
0AH	4	プログラムセグメントプレフィックスのオフセット 6CH の FCB のセグメントアドレス (先の 2 バイト) とオフセットアドレス (続く 2 バイト)。

プロセス中のオープンされたすべてのファイルは、新しくロードされたプログラムでも使用できます。標準入力、標準出力、外部装置、プリンタの各デバイスの細部にわたる情報も親プログラムから引き継がれます。

実行環境 (環境変数 (たとえば、VERIFY=ON) が与える ASCIZ 文字列) も親プロセスから渡されます。環境はパラグラフ (16 の倍数) の境界から始まり、1 バイトの 0 (ASCIZ スtringの終わりも含めて 2 バイトの 00 H) で終わる 32 K バイト未満の ASCIZ 文字列です。最後の 1 バイトの 0 には、プログラムに渡す引数のワードカウントと、引数を表す一連の ASCIZ 文字列が続きます。

カレントディレクトリ中にファイルが見つかった場合、ASCIZ 文字列はファンクション 4 BH から渡される実行可能なプログラムのドライブ名とパス名を含んでいます。ファイルが設定されたパス中で見つかった場合、ファイル名はパス情報 (プログラムをロードするときにこのエリアを使用します) を加えられたものになります。実行環境アドレスが 0 の場合、子プロセスは親プロセスの環境を変化させないで引き継ぎます。

環境のセグメントアドレスを新しいプログラムセグメントプレフィックスのオフセット 2CH に置きます。ロードしたプログラムのために、パラグラフの境界を設定し、パラメータブロックの最初の 2 バイトに環境のセグメントアドレスを置きます。親の環境を受け継いだ場合、パラメータブロックの最初の 2 バイトはともに 0 になります。

COMMAND.COM による子プロセスの起動

COMMAND.COM はつぎの事項を詳細に管理しています。

パス名の設定

プログラムファイルをコマンドパスを通じて検索する

EXE 形式のプログラムを再配置する

そのため、他のプログラムを簡単にロードし、実行する方法として、COMMAND.COM による子プロセスのロードと実行(起動)があります。その方法をつぎに示します。

/C スイッチを含んだコマンドラインを子プロセスに渡し (/C 以下のコマンドラインで子プロセスになるプログラムについて知らせます)、COM 形式、または EXE 形式のプログラムを起動します。

/C スイッチをとともうコマンドラインのフォーマットはつぎのとおりです。

<長さ>/C <コマンド>< 0DH >

<長さ>は、最後のキャリッジリターン (0DH) を含まないコマンドラインの長さです。

<コマンド>は、有効な MS-DOS のコマンドです。

<0DH>は、キャリッジリターンキャラクタです。

プログラムが直接他のプログラムを実行する場合 (COMMAND.COM の代わりに、ファンクション 4BH を使う他のプログラムを指定した場合)、COMMAND.COM が行うすべての作業をアプリケーションで行わなければなりません。

エラーが起きた場合、キャリーフラグが 1 になり、AX にエラーコードが返されます。

エラーリターン

AX

01 H=AL に渡されたファンクションが、00 H ではありません

02 H=指定されたパスが無効であったか、または存在しません。

04 H=現在のプロセスで使用可能な解放されたハンドルが存在しません。

05 H=アクセスが否定されました。

08 H=作成すべきプロセスのための十分な大きさのメモリが、存在しません。

0AH=環境が 32 K バイトを超えています。

0BH=DS:DXによってポイントされているファイルは、EXE形式ファイルで、内部に矛盾した情報が入っています。

マクロ定義

```
exec      macro      path, command, parms
          mov        dx, offset path
          mov        bx, offset parms
          mov        word ptr parms[02H], offset command
          mov        word ptr parms[04H], cs
          mov        word ptr parms[06H], 5CH
          mov        word ptr parms[08H], es
          mov        word ptr parms[0AH], 6CH
          mov        word ptr parms[0CH], es
          mov        al, 0
          mov        ah, 4BH
          int        21H
          endm
```

例

つぎのプログラムは COMMAND.COM をロードし、/W スイッチを使用して DIR コマンドを実行します。

```
pgm_file  db          "command.com", 0
cmd_line  db          9, "/c dir /w", 0DH
parm_blk  db          14 dup (?)
reg_save  db          10 dup (?)
;
func_4B00H set_block  last_inst      ; 割り当てられたブロックの変更(4AH)
exec      pgm_file, cmd_line, parm_blk, 0 ; プログラムをロードし実行
```

INT 21H

Load Overlay

ファンクション 4BH
コード 03H**機能** プログラムセグメント（オーバーレイ）のロード**コール** AH = 4BH

AL = 03H

DS : DX パス名の位置

ES : BX パラメータブロックの位置

リターン キャリーがセットされた場合

AX = 01H 無効なファンクション

02H ファイルが存在しない

04H オープンされているファイルが多すぎる

05H アクセスの否定

0AH 不正な環境

キャリーがセットされない場合

エラーなし

解説

DX は、指定されたプログラムファイルのドライブ名とパス名を表す ASCIZ 文字列のオフセットアドレス（セグメントアドレスは DS）でなければなりません。BX はパラメータブロックのオフセットアドレス（セグメントアドレスは ES）でなければなりません。AL は 03H でなければなりません。

MS-DOS はロードするプログラムが、そのプログラム内にロードする領域を持っているとみなすため、特にメモリを解放（ファンクション 4AH を使って）する必要はありません。また、プログラムセグメントプレフィクスは作成されません。

つぎに、パラメータブロックのアドレスの内容を示します。

オフセット	バイト長	意味
00H	2	プログラムがロードされるセグメントアドレス
02H	2	リロケーション要素、通常、これはパラメータブロックの最初のワード（2 バイト）と同じです。EXE 形式のプログラムとリロケーションの詳細については、第 5 章を参照してください。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

- 01 H=AL に渡されたファンクションが 03 H ではありません。
- 02 H=指定されたパスが無効であったか、ファイルが存在しません。
- 04 H=現在のプロセスで使用可能な解放されたハンドルが存在しません。
- 05 H=アクセスが否定されました。
- 0AH=環境が 32 K バイトを超えています。

マクロ定義

```
exec-ovl      macro      path, parms, seg-addr
               mov        dx, offset path
               mov        bx, offset parms
               mov        parms, seg-addr
               mov        parms[02H], seg-addr
               mov        al, 3
               mov        ah, 4BH
               int         21H
               endm
```

例

つぎのプログラムは、リダイレクトの標準入力として“TEXTFILE.ASC”というファイルをオープンし、オーバーレイとして、“BIT.COM”をロードします。つぎに、“BIT.COM”をコールします。“BIT.COM”は、標準入力として“TEXTFILE.ASC”を読み込みます。

```
stdin      equ      0
;
file        db        "TEXTFILE.ASC", 0
cmd_file    db        "¥bit.com", 0
parm_blk    dw        4 dup (?)
overlay     label     dword
             dw        0
handle      dw        ?
new_mem     dw        ?
;
func_4B03H : set_block last_inst      ; 割り当てられたメモリブロックの変更(4AH)
             jc        setblock_error
```

```

allocate_memory 2000      ;メモリを割り当てる(48H)
jc               allocate_error
mov              new_mem, ax  ;メモリのセグメントアドレスをセーブ
open_handle     file, 0      ;ハンドルのオープン
jc               open_error
mov              handle, ax   ;ハンドルをセーブ
xdup2           handle, stdin ;ファイルのハンドルを二重化(45H)
jc               dup2_error
close_handle     handle      ;ハンドルのクローズ(3EH)
jc               close_error
mov              ax, new_mem   ;新規メモリのアドレスをセット
exec_ovl        cmd_file, parm_blk, ax ;オーバーレイとして
                                   ;プログラムをロード

jc               exec_error
call            overlay        ;オーバーレイをコール
free_memory     new_mem       ;割り当てられたメモリの解放
jc               free_error

```

INT 21H

End Process

ファンクション 4CH

機能	プロセスの終了
コール	AH = 4CH AL リターンコード
リターン	なし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

解説 プロセスを終了し、MS-DOS に制御を戻します。AL は、ファンクション 4DH (子プロセスからリターンコードを得る) で親プロセス、または ERRORLEVEL を使った MS-DOS の IF コマンドから返されるリターンコードでなければなりません。

MS-DOS は、すべてのオープンしているハンドルをクローズし、現在のプロセスを終了します。さらに、制御を起動したプロセスに戻します。

このファンクションは、プログラムセグメントプレフィックスのセグメントアドレスを CS にセットする必要はありません。V 2.0 以前の MS-DOS と互換性を保つ必要がある場合 (割り込みタイプ 20H、ロケーション 0 へのジャンプ) 以外、プログラムの終了には、このファンクションを使用してください。

マクロ定義	end_process	macro	return_code
		mov	al, return_code
		mov	ah, 4CH
		int	21H
		endm	

例

つぎのプログラムはメッセージを表示し、リターンコード 8 で MS-DOS に制御を戻します。このプログラムのメインルーチンはサンプルプログラムを参照してください。

```
message      db      "Displayed by FUNC_4CH example", 0DH, 0AH, "$"
;
func_4CH:    display      message          ; message をスクリーンに出力(09H)
              end_process 8                ; リターンコード 8 でプロセスを終了
code
ends
end          code
```

INT 21H

Get Return Code Child Process

ファンクション 4DH

機能 子プロセスからリターンコードを得る

コール AH = 4DH

リターン AX 抜け出しコード

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 ファンクション 31H (キーププロセス), またはファンクション 4CH (プロセスの終了) で子プロセスを終了するときに指定するリターンコードを1回だけ, 返します。コードは, AL に返されます。AH は, プログラムの終了する状態で, つぎのとおりです。

AH	状態
0	終了
1	〈CTRL-C〉による終了
2	ハードエラー
3	在駐したまま終了 (ファンクション 31H)

エラーリターン なし

マクロ定義

```
ret_code      macro
               mov     ah, 4DH
               int     21H
               endm
```

例 返されるコードが, 状況によっていろいろありますので, プログラムは省略します。

INT 21H

Find First File

ファンクション 4EH

機能 最初に一致するファイル名の検索

コール AH = 4EH
DS : DX パス名の位置
CX 属性

リターン キャリーがセットされた場合
AX = 02H ファイルがない
12H これ以上ファイルがない
キャリーがセットされない場合
エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 指定した、またはカレントのディレクトリ上で指定したパス名と最初に一致するエントリを検索します。DX は、パス名を表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) でなければなりません (ワイルドカードを含むことができます)。CX は、ファイルの検索に使われるアトリビュート (属性) でなければなりません。アトリビュート (属性) の詳細については、1.5.6 を参照してください。

アトリビュートフィールドが、隠しファイル、システムファイル、ディレクトリエントリ (02H, 04H, 10H) のいずれかを1つ以上持っている場合、すべての通常のファイルエントリもまた検索されます。ボリュームラベルを除いたすべてのディレクトリエントリを検索するには、アトリビュートバイトに 16H (隠しファイル+システムファイル+ディレクトリエントリ) をセットします。

アトリビュートとパス名の一致するディレクトリエントリを捜し出した場合、現在のディスク転送アドレス (DTA) で示されるバッファには、つぎの値が書き込まれます。

オフセット	長さ	説明
00H	21	予約 ファンクション 4FH (つぎに一致するファイル名の検索) 用
15H	1	アトリビュートの一致

16H	2	ファイルが最後に書き込まれた時刻
18H	2	ファイルが最後に書き込まれた日付
1AH	2	ファイルサイズの下位ワード (2 バイト)
1CH	2	ファイルサイズの上位ワード (2 バイト)
1EH	13	ファイル名、区切り記号としてのピリオド、拡張子、00H からなります。空白は詰められますので、拡張子がある場合は、ピリオドによって区切られます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

02H = DS : DX 内で指定されたパス名は、無効なパス名です。

12H = 指定したパス名に一致するファイルがありません。

マクロ定義

```
find_first_file    macro        path, attrib
                   mov          dx, offset path
                   mov          cx, attrib
                   mov          ah, 4EH
                   int          21H
                   endm
```

例

つぎのプログラムは、メッセージを表示し、ドライブ B のディスクのカレントディレクトリ上に "REPORT. ASM" を検索します。

```
yes          db          "FILE EXISTS.", 0DH, 0AH, "$"
no           db          "FILE DOES NOT EXIST.", 0DH, 0AH, "$"
path         db          "b: report. asm", 0
buffer       db          43 dup (?)
;
func_4EH:    set_dta      buffer                ; ディスク転送アドレスのセット (1AH)
             find_first_file path, 0           ; 最初に一致するファイル名の検索
             jc          error_findfirst
             cmp         al, 12H                ; これ以上ファイルがないか?
             je          not_there              ; はいの時, not_there へ
             display     yes                    ; yes をスクリーンに出力 (09H)
             jmp         return                 ; 処理終了
not_there:   display     no                     ; no をスクリーンに出力 (09H)
```

INT 21H

Find Next File

ファンクション 4FH

機能 つぎに一致するファイル名の検索

コール AH = 4FH

リターン キャリーがセットされた場合
 AX = 12H これ以上ファイルがない。
 キャリーがセットされない場合
 エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 先のファンクション(先に実行されたファンクション 4EH)で指定されたファイル名を、続けて検索します。現在のディスク転送アドレス(DTA)にはファンクション 4EH、または先行するファンクション 4FH が返したファイル情報が残っていない必要ありません。ファンクション 4EH をコールした後にファンクション 1AH によりディスク転送アドレス(DTA)を変更した場合は、本ファンクションをコールする前にファンクション 4EH をコールしたときのディスク転送アドレス(DTA)に戻さなければなりません。また、ディスク転送アドレス(DTA)の内容はファンクション 4EH を参照してください。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

12H = このパターンと一致するファイルがこれ以上ありません。

マクロ定義	find-next-file	macro	
		mov	ah, 4FH
		int	21H
		endm	

例

つぎのプログラムは、ドライブ B 上のカレントディレクトリ中のすべてのファイル数を表示します。

```

message      db      "No files", 0DH, 0AH, "$"
files        dw      ?
path         db      "b:*. *", 0
buffer       db      43 dup (?)
;
func_4FH:    set_dta   buffer          ; ディスク転送アドレスのセット(1AH)
             find_first_file path, 0   ; 最初に一致するファイル名の検索(4EH)
             jc        error_findfirst
             cmp       al, 12H         ; これ以上ファイルがないか?
             je        all_done        ; はいの時, all_done へ
             inc       files          ; いいえの時, ファイルカウンタを
                                     ; インクリメント
search_dir:   find_next_file          ; 次に一致するファイル名の検索
             jc        error_findnext
             cmp       al, 12H         ; これ以上エントリがあるか?
             je        done           ; いいえの時, done へ
             inc       files          ; はいの時, ファイルカウンタを
                                     ; インクリメント
             jmp       search_dir      ; そして再びチェック
done:         convert   files, 10, message ; 章末参照
all_done:     display   message        ; message をスクリーンに出力(09H)

```


INT 21H

Get Verify State

ファンクション 54H

機 能	ベリファイの状態を得る
コール	AH = 54H
リターン	AL 現在のベリファイフラグの値

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

- 解 説** MS-DOS のディスクファイルへの書き込み時の検証の有無を返します。そのステータスは AL に返され、0 ならばオフ、1 ならばオンです。
- ベリファイフラグの設定については、ファンクション 2EH を参照してください。

マクロ定義

```

get-verify      macro
                mov     ah, 54H
                int     21H
                endm

```

例 つぎのプログラムは、ベリファイのステータスを表示します。

```

message      db      "Verify", "$"
on           db      "on.", 0DH, 0AH, "$"
off          db      "off.", 0DH, 0AH, "$"
;
func_54H:    display  message          ; message をスクリーンに出力(09H)
              get-verify                ; ベリファイの状態を得る
              cmp     al, 0              ; フラグはオフか?

```

	jg	ver_on	; いえの時, ver_on へ
	display	off	; off をスクリーンに出力(09H)
	jmp	return	; 処理終了
ver_on:	display	on	; on をスクリーンに出力(09H)

INT 21H

Change Directory Entry

ファンクション 56H

機 能 ディレクトリエントリの変更

コール AH = 56H

DS : DX 既存のファイルのパス名の位置

ES : DI 新規のパス名の位置

リターン キャリーがセットされた場合

AX = 02H ファイルが存在しない

05H アクセスの否定

11H 装置の不一致

キャリーがセットされない場合

エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説

ディレクトリエントリを変更することによって、ファイル名を変更します。DX は変更されるエントリのパス名の ASCIZ 文字列を表すオフセットアドレス (DS は、セグメントアドレス) です。DI は、変更後のエントリのパス名のオフセットアドレス (ES は、セグメントアドレス) です。

ディレクトリが異なっても、他のディレクトリ上のファイルに変更できます。しかし、ディスクドライブが異なる場合は、変更できません。

このファンクションは、隠しファイル、システムファイル、サブディレクトリを変更することはできません。エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

02H = DS : DX で指定されたファイル名は、存在しません。

05H = DS : DX で指定されたパス名がディレクトリであったか、ES : DI で指定されたファイルがすでに存在しているか、または宛先ディレクトリエントリを作成することができませんでした。

11H = 既存のパスと新規のパスは、異なったドライブに存在します。

マクロ定義	rename_file	macro	old_path, new_path
		mov	dx, offset old_path
		push	ds
		pop	es
		mov	di, offset new_path
		mov	ah, 56H
		int	21H
		endm	

例

つぎのプログラムは、変更前と変更後のファイル名を表示し、ファイル名を変更します。

```

prompt1    db    "Filename: $"
prompt2    db    "New name: $"
old_path    db    15, ?, 15 dup (?)
new_path    db    15, ?, 15 dup (?)
crlf       db    0DH, 0AH, "$"
;
func_56H:   display    prompt1        ; prompt1 をスクリーンに出力(09H)
            get_string  15, old_path    ; 変更前パス名をキーボード入力(0AH)
            xor         bx, bx          ; BL はインデックスとして使用
            mov         bl, old_path[1] ; スtring長を得る
            mov         old_path[bx+2], 0 ; ASCIZ スtringを作成
            display     crlf            ; crlf をスクリーンに出力(09H)
            display     prompt2        ; prompt2 をスクリーンに出力(09H)
            get_string  15, new_path    ; 変更後パス名をキーボード入力(0AH)
            xor         bx, bx          ; BL はインデックスとして使用
            mov         bl, new_path[1] ; スtring長を得る
            mov         new_path[bx+2], 0 ; ASCIZ スtringを作成
            display     crlf            ; crlf をスクリーンに出力(09H)
            rename_file old_path[2], new_path[2] ; ディレクトリエントリを変更
            jc          error_rename

```

INT 21H

Get/Set Date/Time of File

ファンクション 57H

機 能 ファイルの日付／時刻を得る／セットする

コール AH = 57H

AL = 00H 日付／時刻を得る。

01H 日付／時刻をセットする。

BX ファイルハンドル

CX (AL = 01 の場合) セットすべき時刻

DX (AL = 01 の場合) セットすべき日付

リターン キャリーがセットされた場合

AX = 01H 無効なファンクション

06H 無効なハンドル

キャリーがセットされない場合

AL = 01H エラーなし

AL = 00H CX/DX に最後に編集された日時

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 ファイルが最後に編集された日付と時刻をセット、または得ます。日付と時刻を得る場合は、AL は 00H でなければなりません。この時、CX と DX には、時刻と日付がそれぞれ返されます。日付と時刻をセットする場合、AL は 01H でなければなりません。この時、CX と DX は、それぞれ時刻と日付でなければなりません。BX はファイルハンドルでなければなりません。この日付と時刻については、1.8.1 を参照してください。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードが返されます。

エラーリターン

AX

01H = AL に渡されたファンクションは、0～1 の範囲外です。

06H = BX に渡されたハンドルは、現在オープンされていません。

マクロ定義

```

get-set-date-time macro      handle, action, time, date
                           mov      bx, handle
                           mov      al, action
                           mov      cx, word ptr time
                           mov      dx, word ptr date
                           mov      ah, 57H
                           int      21H
                           endm

```

例

つぎのプログラムは、ドライブ B のディスク上の “REPORT.ASM” を得て、その日を翌日に更新し（変更される日付がまたがっている場合、年と月も更新されます）、新しい日付をファイルにセットします。

```

month      db      31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
path       db      "b: report. asm", 0
handle     dw      ?
time       db      2 dup (?)
date       db      2 dup (?)
;
func_57H:  open_handle path, 0          ; ハンドルのオープン(3DH)
           mov      handle, ax          ; ハンドルのセーブ
           get-set-date-time handle, 0, time, date ; ファイルの日付/時刻を得る
           jc      error_time
           mov      word ptr time, cx   ; 時刻をセーブ
           mov      word ptr date, dx   ; 日付をセーブ
           convert_date date[-24]       ; 章末参照
           inc      dh                  ; 日をインクリメント
           xor      bx, bx              ; BL はインデックストとして使用
           mov      bl, dl              ; 月を得る
           cmp      dh, month[bx-1]     ; 月の最終日を越えているか?
           jle      month_ok            ; いいえの時, month_ok へ
           mov      dh, 1               ; はいの時, 日に 1 をセット
           inc      dl                  ; 月をインクリメント
           cmp      dl, 12              ; 月は 12 を越えているか?
           jle      month_ok            ; いいえの時, month_ok へ

```



```
mov     dl, 1           ; はいの時, 月を1にセット
inc     cx              ; 年をインクリメント
month-ok: pack_date     date      ; 章末参照
get-set-date-time handle, 1, time, date ; ファイルの日付/時刻を得る
jc      error-time
close-handle handle      ; ハンドルのクローズ(3EH)
jc      error-close
```

INT 21H

Get/Set Allocation Strategy

ファンクション 58H

機能 アロケーションストラテジを得る／セットする

コール AH = 58H
 AL = 00H ストラテジを得る
 01H ストラテジをセットする
 AL = 01H の場合
 BX = 00H 下位
 01H 最小
 02H 上位

リターン キャリーフラグがセットされている場合
 AX = 01H 無効なファンクションコード
 キャリーフラグがセットされていない場合
 (AL = 00H)
 AX = 00H 下位
 01H 最小
 02H 上位

解説 AL が 00H の場合、AX にストラテジを返します。AL が 01H の場合、BX はストラテジでなければなりません。つぎにストラテジの状態を示します。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

値	名前	意味
00H	下位	MS-DOS はデフォルトとして、最も下位の利用可能なブロックから探し始め、最初に見つかったブロックを割り当てます(割り当てられたメモリは、最も下位の利用可能なブロック)。
01H	最小	MS-DOS は、利用可能な各ブロックを捜し、必要最小の利用可能なブロックを割り当てます。
02H	上位	MS-DOS は、最も上位の利用可能なブロックから探し始め、最初に見つかったブロックを割り当てます(割り当てられたメモリは、最も上位の利用可能なブロック)。

このファンクションリクエストは、MS-DOS のメモリの管理を制御できます。
エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーコード

AX

01H=AL が 00H か 01H ではない、または BX が 00H~02H ではない。

マクロ定義

```
alloc_strat      macro      code, strategy
                  mov        bx, strategy
                  mov        al, code
                  mov        ah, 58H
                  int        21H
                  endm
```

例

つぎのプログラムは、実際のメモリアロケーションストラテジを表示し、ストラテジを上位(2)にセットすることにより、つぎに割り当てられるメモリを利用可能なメモリの最も上位のものにします。

```
get      equ      0
set      equ      1
stdout   equ      1
last_fit equ      2
;
first    db        "First fit ",0DH, 0AH
best     db        "Best fit ",0DH, 0AH
last     db        "Last fit ",0DH, 0AH
;
func_58H: alloc_strat get          ; アロケーションストラテジを得る
          jc        alloc_error
          mov       cl, 4          ; オフセットを算出するために
          shl       ax, cl         ; リターンコードを16倍する
          mov       dx, offset first ; first メッセージのオフセットをセット
          add       dx, ax         ; そしてベースアドレスを加算
          mov       bx, stdout     ; 書き込むハンドルを指定
          mov       cs, 16         ; 16バイト書き込む
```



```
mov      ah, 40h          ; ファンクションコードを指定
int      21H              ; システムコール(40H)
jc       write_error      ; 2 Error 3 4 5
alloc_strat set, last_fit  ; アロケーションストラテジをセット
jc       alloc_error
```

INT 21H

Get Extended Error

ファンクション 59H

機能 拡張されたエラーコードを得る

コール AH = 59H
BX = 0

リターン AX 拡張されたエラーコード
BH エラークラス
BL 可能な対処
CH ローカス
CL, DX, SI, DI, BP, DS, ES の各レジスタの内容は破壊されます。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説 MS-DOS の機能の拡張 (MS-DOS V2.0 と比較して) に対して、新しい機能に対応するために、エラーコードを拡張します。このファンクションを実行すると、CL, DX, SI, DI, BP, DS, ES の各レジスタの内容は破壊されます。

ユーザーが用意した割り込みタイプ 24H のハンドラで、このファンクションを使うと致命的エラーについての詳細な情報を得ることができます。

コールの BX はエラーのレベルを表します。通常は 00H です。

このファンクションの 4 つのリターン情報 (AX, BH, BL, CH の 4 つのレジスタに返される) の詳細についてつぎに示します (AX については、エラーコード一覧を参照してください)。

BH エラークラス

BH は、エラーのクラスに関するコードを返します。つぎにその内容を示します。

コード 意味

01H メモリ容量や I/O チャンネルなどの資源の不足

02H エラーではありませんが、終了するべき一時的状況 (ファイルの一部がロックされている) に陥っています。

03H 権限の問題

04H	システムソフトウェアの内部エラー
05H	ハードウェアに起因するエラー
06H	現在のプロセスが原因でないシステムソフトウェアのエラー
07H	アプリケーションプログラムのエラー
08H	ファイルまたは項目がありません。
09H	ファイルまたは項目が無効なフォーマットまたはタイプです。またそうでない場合は、ファイルまたは項目が無効か、適切ではありません。
0AH	ファイルまたは項目が内部的にロックされています。
0BH	ドライブ内のディスク上に問題があります。ディスクの一部分か、記憶媒体自身に問題があります。
0CH	その他の原因によるエラー

BL 可能な対処

BL は、エラーに対してプログラムが対応できることを示唆するコードを返します。

コード 意味

01H	再試行、ユーザーに確認を求める。
02H	休止後に再試行。
03H	ドライブ名やファイル名などのデータの入力の場合、ユーザーに再度の入力を求めます。
04H	メモリの内容をクリアして、終了します。
05H	すぐに終了してください。ファイルのクローズやインデックスのアップデートよりも優先して、すぐにプログラムが終了しなければならないほど、システムの状況が異常です。
06H	エラーコードを参考にしてください。
07H	ディスクを取り換え、再試行するなどの動作をユーザー側で行わなければなりません。

CH ローカス

CH は、エラーにともなうメモリの種類などの付加情報のコードを返します。これらは、特にハードウェアに起因するエラーです (BH=5)

コード 意味

01H	不明
02H	ディスクドライブのような、ランダムアクセスブロックデバイスに関するエラーです。

- 03H ネットワークに関するエラーです。
- 04H プリンタのような、シリアルアクセスキャラクタデバイスに関するエラーです。
- 05H ランダムアクセスメモリ (RAM) に関するエラーです。

プログラムでは、V 3.0 以前のシステムコールでエラーが起きると、このシステムコールを実行します。これによって拡張されたエラーコードを得ることができます。プログラムが拡張されたエラーコードを使わなくても、V3.0 以前のエラーコードで対応できます。

このシステムコールは、割り込みタイプ 24H で利用でき、ネットワーク関係のエラーコードを返すことができます。

マクロ定義

```
get_error      macro
                mov     ah, 59H
                mov     bx, 0
                int     21H
                endm
```

例

このファンクションリクエストは、割り込みなどの種々の状況を設定しなければならぬのでプログラムは省略します。

INT 21H

Create Temporary File

ファンクション 5AH

機 能	一時ファイルの作成
コール	AH = 5AH CX アトリビュート DS:DX 1バイトの 00H とメモリの 13 バイト が続くパス名の位置
リターン	キャリーフラグがセットされている場合 AX = 03H パス名がない 05H アクセスできない キャリーフラグがセットされない場合 AX ファイルハンドル

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGS _H		FLAGS _L
CS		
DS		
SS		
ES		

解 説 DX は、パス名、00H とメモリの 13 バイト（ファイル名を保持している）からなる ASCIZ 文字列のオフセットアドレス（セグメントアドレスは、DS）でなければなりません。CX は、ファイルに割り当てられたアトリビュートでなければなりません。アトリビュートについては 1.5.6 を参照してください。

MS-DOS は、特別なファイル名を作成し、そのファイル名に DS:DX が指定するパス名を付け加えます。つぎに、そのファイルを作成し、通常のファイルと互換性のあるモードでオープンし、AX にファイルハンドルを返します。一時ファイルを必要とするプログラムは、このファンクションを使って、重複したファイル名を使用しないようにします。

MS-DOS は、作成したプロセスが終了したときに、ファンクション 5AH を使って作成したファイルを自動的に消去しません。ファイルが必要でなくなった時点で消去してください。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

03H=DS:DX で指定したディレクトリが無効か、存在しません。

05H=アクセスできません。

マクロ定義

```

create-temp    macro    pathname, attrib
                mov      cx, attrib
                mov      dx, offset pathname
                mov      ah, 5AH
                int       21H
                endm

```

例

つぎのプログラムは、ディレクトリ “¥WP¥DOCS” に一時ファイルを作成し、カレントディレクトリの “TEXTFILE. ASC” を一時ファイル内にコピーし、両方のファイルをクローズします。

```

stdout        equ      1
;
file           db        "TEXTFILE. ASC", 0
path           db        "¥WP¥DOCS", 0
temp           db        13 dup (0)
open_msg       db        "opened.", 0DH, 0AH
crl_msg        db        "created.", 0DH, 0AH
rd_msg         db        "read into buffer.", 0DH, 0AH
wr_msg         db        "Buffer written to"
cl_msg         db        "Files closed.", 0DH, 0AH
crlf           db        0DH, 0AH
handle1        dw        ?
handle2        dw        ?
buffer         db        512 dup (?)
;
func_5AH:      open_handle file, 0          ; ハンドルのオープン(3DH)
                jc         open_error
                mov        handle1, ax      ; ハンドルのセーブ
                write_handle stdout, file, 12 ; file を表示(40H)
                jc         write_error
                write_handle stdout, open_msg, 10 ; open_msg を表示(40H)
                jc         write_error
                create_temp path, 0         ; 一時ファイルを作成

```



```

jc          create_error
mov         handle2, ax          ; ハンドルをセーブ
write_handle stdout, path, 8     ; path を表示(40H)
jc          write_error
display_char "¥"                ; 文字"¥"を表示(02H)
write_handle stdout, temp, 12    ; temp を表示(40H)
jc          write_error
write_handle stdout, crl_msg, 11 ; crl-msg を表示(40H)
jc          write_error
read_handle handle1, buffer, 512 ; ハンドルで指定されたファイル
                                   ; から読み込む(3FH)
jc          read_error
write_handle stdout, file, 12    ; file を表示(40H)
jc          write_error
write_handle stdout, rd_msg, 20  ; rd-msg を表示(40H)
jc          write_error
write_handle handle2, buffer, 512 ; ハンドルで指定されたファイル
                                   ; に書き込む(40H)
jc          write_error
write_handle stdout, wr_msg, 18  ; wr-msg を表示(40H)
jc          write_error
write_handle stdout, temp, 12    ; temp を表示(40H)
jc          write_error
write_handle stdout, crlf, 2     ; crlf を表示(40H)
jc          write_error
close_handle handle1            ; ハンドルのクローズ
jc          close_error
close_handle handle2            ; ハンドルのクローズ
jc          close_error
write_handle stdout, cl_msg, 15  ; cl_msg を表示(40H)
jc          write_error

```

INT 21H

Create New File

ファンクション 5BH

機 能 新しいファイルの作成

コール AH = 5BH

CX アトリビュート

DS: DX パス名の位置

リターン キャリーフラグがセットされている場合

AX = 03H パスが存在しない

04H オープンするファイル数が多すぎる

05H アクセスできない

50H ファイルが既に存在する。

キャリーフラグがセットされていない場合

AX ファイルハンドル

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説 DX は、パス名を表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) でなければなりません。CX は、アトリビュートでなければなりません (アトリビュートの詳細は 1.5.6 を参照してください)。

同じファイル名が存在しない限り、MS-DOS はファイルを作成し、V2.0 と互換性のあるモードでオープンし、AX にハンドルを返します。

ファンクション 3CH (ハンドルの作成) は、同じファイル名が存在すると、ファイルの内容が 0 バイトのファイル名を作成していますが、このファンクションは、エラーを返します。また、ファイルの存在は、マルチタスクシステムのセマフォとして使えますので、このシステムコールはセマフォのテストとセットに使用できます。

エラーが起きると、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

03H=DS:DX で指定したディレクトリが無効または存在しない。

04H=カレントプロセス中に、利用可能なハンドルがないか、MS-DOS内部システムテーブルが一杯です。

05H=アクセスできない。

50H=DS:DX で指定したファイル名が既に存在します。

マクロ定義

```
create_new      macro      pathname, attrib
                mov        cx, attrib
                mov        dx, offses pathname
                mov        ah, 5BH
                int        21H
                endm
```

例

つぎのプログラムは、カレントディレクトリに“REPORT.ASM”という名の新しいファイルを作成します。同じ名のファイルが存在した場合、エラーメッセージを表示し、MS-DOSに戻ります。同じ名のファイルが存在せず、他のエラーがない場合、プログラムはハンドルをセーブし、プロセスを続行します。

```
err_msg      db      "FILE ALREADY EXISTS", 0DH, 0AH, "$"
path         db      "REPORT.ASM", 0
handle       dw      ?
;
func_5BH:    create_new path, 0          ; 新しいファイルを作成
            jnc      continue          ; エラーのない時、プロセスを実行
            cmp      ax, 80            ; ファイルは既に存在するか?
            jne      error
            display  err_msg          ; err_msg をスクリーンに出力(09H)
            jmp      return           ; MS-DOSに戻る
continue:    mov      handle, ax       ; ハンドルのセーブ
;
;      (further processing here)
```


INT 21H

Lock

ファンクション 5CH
コード 00H**機能** ファイルアクセスのロック**コール** AH = 5CH

AL = 00H

BX ファイルハンドル

CX:DX ロックされた領域のオフセット

SI:DI ロックされた領域の長さ

リターン キャリーフラグがセットされた場合

AX = 01H 無効なファンクションコード

06H 無効なハンドル

21H ロックの破壊

キャリーフラグがセットされない場合

エラーなし

解説

BX は、ロックされた領域を含むファイルのハンドルでなければなりません。CX:DX (4 バイト整数) は、ファイル内のロックされた領域の始めのオフセットでなければなりません。SI:DI (4 バイト整数) は、領域の長さでなければなりません。

他のプロセスがロックされた領域にアクセス（読み出しか書き込み）を行おうとすると、MS-DOS は 3 回再試行し、失敗するとそのプロセスのために割り込みタイプ 24H を実行します。再試行の回数の変更については、ファンクション 44H、コード 0BH を参照してください。

ロックされた領域は、ファイル中のどこかにありうるわけです。その領域が EOF を超えていてもエラーにはならず、領域は短期間ロックされており、一定時間（ハードウェアに依存します。たとえば 10 秒）以上たってもなお領域がロックされている場合にはエラーになります。

ファンクション 45H（ファイルハンドルの二重化）と 46H（ファイルハンドルの強制二重化）は、ロックされた領域に関してもアクセスします。ファンクション 4BH、コード 00H（プログラムのロードと実行）を使って、子プロセスにオープンファイルを渡してもロックされた領域に多重アクセスすることはできません。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

プログラムがロックされた領域を含むファイルを閉じるか、またはロックされた領域を含むファイルをオープンしたまま終了した場合、結果は保証されません。割り込みタイプ 23H (CTRL-C)、24H (致命的エラー) によって終了するプログラムは、割り込みタイプを回避するか、または終了する前にロックされた領域をアンロック (解除) します。

プログラムはロックされた領域がアクセスできないことを認識できません。領域をロックしようとしてエラーコードを確認することによって、プログラムは、領域のステータス (ロックされているか否か) を確認することができます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = このファンクションリクエストを使うためには、ファイルシェアリング (SHARE.EXE) がロード (常駐) されていなければなりません。

06H = BX のハンドルが無効か、オープンされたハンドルです。

21H = 領域のすべてか一部がロックされています。

マクロ定義

```
lock      macro      handle, start, bytes
          mov        bx, handle
          mov        cx, word ptr start
          mov        dx, word ptr start+2
          mov        si, word ptr bytes
          mov        di, word ptr bytes+2
          mov        al, 0
          mov        ah, 5CH
          int        21H
          endm
```

例

つぎのプログラムは、ロックされていない "FINALRPT" という名のファイルをオープンし、最初の 128 バイトと 1024 バイトから 5116 バイトまでの 2 個所をロックします。いくつかの作業の後に、同じ場所をアンロックし、クローズします。

```

    stdout      equ      1
;
    start1      dd      0
    lgth1       dd      128
    start2      dd      1023
    lgth2       dd      4096
    file        db      "FINALRPT", 0
    op_msg      db      "opened.", 0DH, 0AH
    l1_msg      db      "First 128 bytes locked.", 0DH, 0AH
    l2_msg      db      "Bytes 1024-5119 locked.", 0DH, 0AH
    u1_msg      db      "First 128 bytes unlocked.", 0DH, 0AH
    u2_msg      db      "Bytes 1024-5119 unlocked.", 0DH, 0AH
    cl_msg      db      "closed.", 0DH, 0AH
    handle      dw      ?
;
func_5C00H: open_handle file, 01000010b    ; ハンドルのオープン(3DH)
            jc          open_error
            write_handle stdout, file, 8    ; file を表示(40H)
            jc          write_error
            write_handle stdout, op_msg, 10 ; op_msg を表示(40H)
            jc          write_error
            mov         handle, ax          ; ハンドルをセーブ
            lock        handle, start1, lgth1 ; ファイルアクセスのロック
            jc          lock_error
            write_handle stdout, l1_msg, 25 ; l1_msg を表示(40H)
            jc          write_error
            lock        handle, start2, lgth2 ; ファイルアクセスのロック
            jc          lock_error
            write_handle stdout, l2_msg, 25 ; l2_msg を表示
            jc          write_error
;
; (Further processing here)
;
            unlock     handle, start1, lgth1 ; ファイルアクセスのロックを
                                           ; 解除(5C01H)

```



```

jc          unlock_error
write_handle stdout, u1_msg, 27 ; u1_msg を表示(40H)
jc          write_error
unlock      handle, start2, lgth2 ; ファイルアクセスのロックを
                                         ; 解除(5C01H)

jc          unlock_error
write handle stdout, u2_msg, 27 ; u2_msg を表示(40H)
jc          write_error
close_handle handle          ; ハンドルのクローズ(3EH)
jc          close_error
write_handle stdout, file, 8   ; file を表示(40H)
jc          write_error
write_handle stdout, cl_msg, 10 ; cl_msg を表示
jc          write_error

```

INT 21H

Unlock

ファンクション 5CH
コード 01H**機能** ファイルアクセスのロック解除**コール** AH = 5CH

AL = 01H

BX ハンドル

CX:DX ロックを解除する領域のオフセット

SI:DI ロックを解除する領域の長さ

リターン キャリーフラグがセットされた場合

AX = 01H 無効なコード

06H 無効なハンドル

21H ロックの破壊

キャリーフラグがセットされない場合

エラーなし

解説

BX は、ロックを解除する領域を含むファイルのハンドルでなければなりません。CX:DX (4 バイト整数) は、ファイル内のロックされた領域の始めのオフセットでなければなりません。SI:DI (4 バイト整数) は、領域の長さでなければなりません。このオフセットと領域の長さは、ファンクション 5CH、コード 00H (ロック) でロックされたときに指定されたものと同じでなければなりません。

ロックされる領域については、ファンクション 5CH、コード 00H (ロック) を参照してください。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = このファンクションリクエストを使うためには、ファイルシェアリング (SHARE.EXE) がロード (常駐) されていなければなりません。

06H = BX のハンドルが無効か、オープンされたハンドルです。

21H = 指定した領域は、ファンクション 5CH、コード 00H でロックされた領域ではありません。

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

マクロ定義

unlock

```
macro      handle, start, bytes
mov        bx, handle
mov        cx, word ptr start
mov        dx, word ptr start+2
mov        si, word ptr bytes
mov        di, word ptr bytes+2
mov        al, 1
mov        ah, 5CH
int        21H
endm
```

例

ファンクション 5CH, コード 00H を参照してください.

INT 21H

Get Machine Name

ファンクション 5EH
コード 00H

機能 マシン名を得る

コール AH = 5EH

AL = 00H

DS:DX 16 バイトのバッファの位置

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクションコード

キャリーフラグがセットされていない場合

CX ローカルコンピュータの番号

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

このファンクションは、ローカルコンピュータのネット名を得ます。DX は、16 バイトのバッファのオフセットアドレス (DS は、セグメントアドレス) でなければなりません。MS-Networks が稼動していなければなりません。

MS-DOS は、DS:DX の指定するバッファ中のローカルコンピュータ名(16 バイトの ASCIZ 文字列、ブランクは詰めます) を返します。CX は、ローカルコンピュータの番号を返します。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H=このファンクションリクエストを実行するには、MS-Networks が稼動していなければなりません。

マクロ定義

```

get_machine_name macro      buffer
                        mov    dx, offset buffer
                        mov    al, 0
                        mov    ah, 5EH
                        int     21H
                        endm

```

例

つぎのプログラムは、MS-Networks のワークステーションの名前を表示します。

```

stdout      equ          1
;
msg          db           "Netname:"
mac_name     db           16 dup (?), 0DH, 0AH
;
func_5E00H   ; get_machine_name mac_name      ; ワークステーションの名前を得る
              jc          name_error
              write_handle stdout, msg, 27    ; msg を表示(40H)
              jc          write_error

```

INT 21H

Printer Setup

ファンクション 5EH
コード 02H

機能 プリンタセットアップ

コール AH = 5EH

AL = 02H

BX 割り当てリストのインデックス

CX セットアップ文字列の長さ

DS:SI セットアップ文字列の位置

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクションコード

キャリーフラグがセットされていない場合

エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

ネットワークプリンタに送る各ファイルの先頭に、MS-DOS が付けるコントロールキャラクタを定義します。BX は、プリンタの割り当てリストの中のインデックス（エントリ 0 は、最初のエントリになります）でなければなりません。CX は、セットアップ文字列の長さでなければなりません。SI は、セットアップ文字列のオフセットアドレス（DS は、セグメントアドレス）でなければなりません。MS-Networks が稼動していなければなりません。

セットアップ文字列は、BX の割り当てリストのインデックスによって、プリンタに送られる各ファイルの先頭に付け加えます。このファンクションリクエストは、プリンタコンフィグレーションを持ったプリンタを受け持つプログラムで使われます。ファンクション 5FH、コール 02H を使って、プリンタの割り当てリストを登録することができます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = このファンクションリクエストを実行するには、MS-Networks が稼動していなければなりません。

マクロ定義

printer_setup

macro

index, lgth, string

mov

bx, index

mov

cx, lgth

mov

dx, offset string

mov

al, 2

mov

ah, 5EH

int

21H

endm

例

各種のプリンタに依存するところなのでプログラムは省略します。

INT 21H

Get Assign List Entry

ファンクション 5FH
コード 02H

機能 割り当てリストのエントリを得る

コール AH = 5FH

AL = 02H

BX 割り当てリストのインデックス

DS:SI ローカル名のバッファの位置

ES:DI リモート名のバッファの位置

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクションコード

12H これ以上のファイルはない

キャリーフラグがセットされていない場合

BL = 03H プリンタ

04H ドライブ

CX ユーザー変数域

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

このファンクションは、ネットワークの割り当てリストのエントリを得ます。BX は、割り当てリストインデックス（エントリ 0 のときは最初のエントリ）でなければなりません。SI は、ローカル名のための 16 バイトのバッファのオフセットアドレス（DS は、セグメントアドレス）でなければなりません。DI は、リモート名の 128 バイトのバッファのオフセットアドレス（ES は、セグメントアドレス）でなければなりません。MS-Networks が稼動していなければなりません。

MS-DOS は、DS:SI で指定するバッファ内のローカル名と ES:DI で指定するバッファ内のリモート名を設定します。ローカル名はヌルの ASCIZ 文字列もとれます。BL は、ローカルデバイスがプリンタの場合は 03H、デバイスの場合は 04H を返します。CX は、ファンクション 5FH、コード 03H（割り当てリストのエントリの作成）でセットされたユーザー変数の値を返します。割り当てリストは、その内容を書き換えることもできます。

このファンクションリクエストを使って、エントリを得るか、またはテーブルを検索して完成したリストのコピーを作ることができます。割り当てリストの終わりを見つけると、ファンクション 4EH（最初に一致するファイル名の検索）、4FH（つぎに一致するファイル名の検索）でディレクトリを検索するときのように、

エラーコード 12H をチェックします。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H = このファンクションリクエストを実行するには、MS-Networks が稼動していなければなりません。

12H = BX で得られたインデックスが、割り当てのリストのエントリの数より大きい。

マクロ定義	get_list	macro	index, local, remote
		mov	bx, index
		mov	si, offset local
		mov	di, offset remote
		mov	al, 2
		mov	ah, 5FH
		int	21H
		endm	

例 つぎのプログラムは、MS-Networks のワークステーションの各エントリのローカル名、リモート名、デバイスタイプ(ドライブかプリンタ)、割り当てリストを表示します。

```

stdout      equ      1
printer     equ      3
;
local_nm    db        16 dup (?), 2 dup (20h)
remote_nm   db        128 dup (?), 2 dup (20h)
header      db        "Local name", 8 dup (20h)
            db        "Remote name", 7 dup (20h)
            db        "Device Type"
crlf        db        0dh, 0ah, 0dh, 0ah
drive_msg   db        "drive"
print_msg   db        "printer"
index       dw        ?
;

```



```

func_5F02H: write_handle stdout, header, 51 ; header を表示(40H)
            jc write_error
            mov index, 0 ; 割り当てリストのインデックスをセット
ck_list:    get_list index, local_nm, remote_nm ; 割り当てリストのエントリを得る
            jnc got_one ; 1 エントリを得る, got_one へ
error:      cmp ax, 18 ; ラストエントリか?
            je last_one ; はいの時, last_one へ
            jmp error
got_one:    push bx ; デバイスタイプをセーブ
            write_handle stdout, local_nm, 148 ; local_nm を表示(40H)
            jc write_error
            pop bx ; デバイスタイプをリストア
            cmp bl, printer ; プリンタデバイスか?
            je prntr ; はいの時, print へ
            write_handle stdout, drive_msg, 5 ; drive_msg を表示(40H)
            jc write_error
            jmp get_next ; get_next へ
prntr:      write_handle stdout, print_msg, 7 ; print_msg を表示(40H)
            jc write_error
get_next:   write_handle stdout, crlf, 2 ; crlf を表示(40H)
            jc write_error
            inc index ; インデックスをインクリメント
            jmp ck_list ; 次のエントリを得る
last_one:   write_handle stdout, crlf, 4 ; crlf を表示(40H)
            jc write_error
;
            jmp return

```

INT 21H

Make Assign List Entry

ファンクション 5FH
コード 03H

- 機能** 割り当てリストのエントリの作成
- コール** AH = 5FH
AL = 03H
BL = 03H プリンタ
04H ドライブ
CX ユーザー変数域
DS:SI ソースデバイス名の位置
ES:DI ディスティネーションデバイス名の位置
- リターン** キャリーフラグがセットされている場合
AX = 01H 無効なファンクションコード
03H パスが見つからない
05H アクセスできない
08H メモリ不足（ネットワークが起
こしたエラーによる）
キャリーフラグがセットされていない場合
エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
SP		
BP		
SI		
DI		
IP		
FLAGSH		FLAGSL
CS		
DS		
SS		
ES		

- 解説** このファンクションは、プリンタまたはディスクドライブ（ソースデバイス）をネットワークディレクトリ（ディスティネーションデバイス）としてリディレクトします。BL は、ソースデバイスがプリンタなら 03H、ディスクドライブなら 04H でなければなりません。
- SI は、プリンタ名、コロン付きのドライブ名、ヌル文字列(1 バイトの 00H)のいずれかを表す ASCIZ 文字列のオフセットアドレス (DS は、セグメントアドレス) でなければなりません。DI は、ネットワークディレクトリ名を表す ASCIZ 文字列のオフセットアドレス (ES は、セグメントアドレス) でなければなりません。CX は、MS-DOS が提供しているユーザーが使える 16 ビットの変数でなければなりません。MS-Networks が稼動していなければなりません。

ディスティネーション文字列は、つぎのような書式でなければなりません。

<マシン名><パス名>< 00H ><パスワード>< 00H >

<マシン名>は、ネットワークのサーバのネット名です。

<パス名>は、ソースデバイスからリディレクトに渡されるネットワークディレクトリのアリアス（別名）です。

<00H>：ヌルコード

<パスワード>は、ネットワークをアクセスするためのパスワードです。パスワードがない場合、<パス名>の後には、2 バイトのヌルコードが続かなければなりません。

BL=03H の場合、ソース文字列は、PRN でなければなりません。プリンタとして登録されたすべての出力はバッファに貯められ、ディスティネーション文字列に登録されたリモートプリンタスプーラに送られます。

BL=04H の場合、ソース文字列は、コロン付きのドライブ名か、ヌル文字列のいずれかでなければなりません。ソース文字列が無効なドライブ名とコロンの場合、それ以降のすべてのドライブ名は、ディスティネーション文字列に登録されたネットワークディレクトリにリディレクトに渡されたものと見なします。ソース文字列がヌルの場合、MS-DOS は、パスワードが合うネットワークディレクトリとしてアクセスしようとしています。

ディスティネーション文字列は、128 バイト以下でなければなりません。CX のユーザー変数は、ファンクション 5FH、コード 02H (割り当てリストエントリを得る) で与えられます。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーリターン

AX

01H=このファンクションリクエストを実行するのに必要な MS-Net works が稼動していない、BX は 01H から 04H の範囲にない、ソース文字列の書式が誤りである、ディスティネーション文字列の書式が誤りである、ソースデバイスがすでにリディレクトに渡されている、のいずれかです。

03H=ネットワークディレクトリが、無効か、または存在しない。

05H=ネットワークディレクトリ/パスワードが有効ではない。パスワードが無効であるか、ディレクトリがサーバ上に存在しない。

08H=文字列の置かれているメモリが十分でない。

マクロ定義	redir	macro	device, value, source, destination
		mov	bl, device
		mov	cx, value
		mov	si, offset source
		mov	es, seg destination
		mov	di, offset destination
		mov	al, 03H
		mov	ah, 5FH
		int	21H
		endm	

例 つぎのプログラムは“HAROLD”という名のサーバに、ワークステーションから、2つのデバイスとプリンタをリディレクトに渡します。マシン名、ディレクトリ名、ドライブ文字は、つぎのようになります。

ローカルのドライブ またはプリンタ	サーバ上のネット名	パスワード
E:	WORD	なし
F:	COMM	fred
PRN:	PRINTER	quick

```

printer      equ      3
drive        equ      4
local_1      db        "e:", 0
local_2      db        "f:", 0
local_3      db        "prn", 0
remote_1     db        "%harold%word", 0, 0
remote_2     db        "%harold%comm", 0, "fred", 0
remote_3     db        "%harold%printer", 0, "quick", 0
func_5F03H:  redir     local_1, remote_1 drive, 0 ; ドライブを E:WORD という名前で
                jc      error                    ; リディレクトに渡す
                redir   local_2, remote_2, drive, 0 ; ドライブを F:COMM という名前で
                jc      error                    ; リディレクトに渡す
                redir   local_3 remote_3 printer, 0 ; プリンタを PRINTER という名前で
                jc      error                    ; リディレクトに渡す

```

INT 21H

Cancel Assign List Entry

ファンクション 5FH
コード 04H**機能** 割り当てリストのエントリのキャンセル**コール** AH = 5FH

AL = 04H

DS:SI ソースデバイスの名前の位置

リターン キャリーフラグがセットされている場合

AX = 01H 無効なファンクションコード

0FH サーバ上のリディレクトの中止

キャリーフラグがセットされていない場合

エラーなし

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解説

このファンクションは、プリンタまたはディスクドライブ（ソースデバイス）の、ファンクション 5FH、コード 04H で作成されたネットワークディレクトリ（ディスティネーションデバイス）へのリディレクトをキャンセルします。SI は、キャンセルするリディレクトのプリンタまたはドライブ名を表す ASCIZ 文字列のオフセットアドレス（DS は、セグメントアドレス）です。MS-Networks が稼動していなければなりません。

DS:SI で指定される ASCIZ 文字列の値は、つぎの3つのいずれかです。

1. リディレクトのコロン付きのドライブ名。リディレクトをキャンセルし、物理的なドライブ名に戻ります。
2. リディレクトのプリンタの名前 (PRN)。リディレクトをキャンセルし、物理的なプリンタ名に戻ります。
3. ¥¥(¥マーク2つ)で始まる文字列。ローカルマシンとネットワークディレクトリの接続が終了したことを示します。

エラーが起きた場合、キャリーフラグがセットされ、AX にエラーコードを返します。

エラーコード

AX

01H=このファンクションを使うには、MS-Networks が稼動していなければなりません。または、ASCIZ 文字列が実在するソースデバイスの名前ではありません。

0FH=ネットワークのリディレクトのディスクまたはプリンタが停止しています。

マクロ定義

```
cancel_redir    macro          local
                 mov           si, offset local
                 mov           al, 4
                 mov           ah, 5FH
                 int           21H
                 endm
```

例

つぎのプログラムは、MS-Networks のドライブ E, F とプリンタ (PRN) のリディレクトをキャンセルします。ただし、これらは、ローカルデバイスとして、前もってリディレクトされていなければなりません。

```
local_1         db            "e:", 0
local_2         db            "f:", 0
local_3         db            "prn", 0
;
func_5F04H:      cancel_redir  local_1          ; ドライブ E のリディレクトをキャンセル
                 jc            error
                 cancel_redir  local_2          ; ドライブ F のリディレクトをキャンセル
                 jc            error
                 cancel_redir  local_3          ; プリンタ PRN のリディレクトをキャンセル
                 jc            error
```


INT 21H

Get PSP

ファンクション 62H

機 能	PSP を得る
コール	AH = 62H
リターン	BX カレントプロセスのプログラムセグメント プレフィックスのセグメントアドレス

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL

SP
BP
SI
DI

IP	
FLAGSH	FLAGSL

CS
DS
SS
ES

解 説	このファンクションは、現在の実働しているプロセスのセグメントアドレス（プログラムセグメントプレフィックスの先頭）を返します。
-----	--

マクロ定義	get-psp	macro
	mov	ah, 62H
	int	21H
	endm	

例	つぎのプログラムは、プログラムセグメントプレフィックスのセグメントアドレスを10進数で表示します。
---	---

```

msg          db          "PSP segment address: H", 0DH, 0AH, "$"
;
func-62H:    get-psp          ; PSP のセグメントアドレスを得る
              convert bx, 16, msg[21] ; 章末参照
              display msg      ; msg をスクリーンに出力(09H)

```

1.12 MS-DOS システムコールにおけるマクロ定義例

```

;*****
; Interrupts
;*****
;
;                                INTERRUPT 25H
abs_disk_read macro disk,buffer,num_sectors,first_sector
    mov     al,disk
    mov     bx,offset buffer
    mov     cx,num_sectors
    mov     dx,first_sector
    int     25H
    popf
endm

;                                INTERRUPT 26H
abs_disk_write macro disk,buffer,num_sectors,first_sector
    mov     al,disk
    mov     bx,offset buffer
    mov     cx,num_sectors
    mov     dx,first_sector
    int     26H
    popf
endm

;                                INTERRUPT 27H
stay_resident macro last_instruc
    mov     dx,offset last_instruc
    inc     dx
    int     27H
endm

;
;
;*****
; Function Requests
;*****
;                                FUNCTION REQUEST 00H
terminate_program macro
    xor     ah,ah
    int     21H
endm

;                                FUNCTION REQUEST 01H
read_kbd_and_echo macro
    mov     ah,01H
    int     21H
endm

;                                FUNCTION REQUEST 02H
display_char macro character
    mov     dl,character
    mov     ah,02H
    int     21H
endm

```

```
;
aux_input macro
    mov     ah,03H
    int     21H
endm
```

FUNCTION REQUEST 03H

```
;
aux_output macro
    mov     ah,04H
    int     21H
endm
```

FUNCTION REQUEST 04H

```
;
print_char macro character
    mov     dl,character
    mov     ah,05H
    int     21H
endm
```

FUNCTION REQUEST 05H

```
;
dir_console_io macro switch
    mov     dl,switch
    mov     ah,06H
    int     21H
endm
```

FUNCTION REQUEST 06H

```
;
dir_console_input macro
    mov     ah,07H
    int     21H
endm
```

FUNCTION REQUEST 07H

```
;
read_kbd macro
    mov     ah,08H
    int     21H
endm
```

FUNCTION REQUEST 08H

```
;
display macro string
    mov     dx,offset string
    mov     ah,09H
    int     21H
endm
```

FUNCTION REQUEST 09H

```
;
get_string macro limit,string
    mov     dx,offset string
    mov     string,limit
    mov     ah,0AH
    int     21H
endm
```

FUNCTION REQUEST 0AH

```
;
check_kbd_status macro
    mov     ah,0BH
    int     21H
endm
```

FUNCTION REQUEST 0BH


```

;                                FUNCTION REQUEST 0CH
flush_and_read_kbd macro switch
    mov     al,switch
    mov     ah,0CH
    int     21H
endm

;                                FUNCTION REQUEST 0DH
reset_disk macro
    mov     ah,0DH
    int     21H
endm

;                                FUNCTION REQUEST 0EH
select_disk macro disk
    mov     dl,disk[-65]
    mov     ah,0EH
    int     21H
endm

;                                FUNCTION REQUEST 0FH
open macro fcb
    mov     dx,offset fcb
    mov     ah,0FH
    int     21H
endm

;                                FUNCTION REQUEST 10H
close macro fcb
    mov     dx,offset fcb
    mov     ah,10H
    int     21H
endm

;                                FUNCTION REQUEST 11H
search_first macro fcb
    mov     dx,offset fcb
    mov     ah,11H
    int     21H
endm

;                                FUNCTION REQUEST 12H
search_next macro fcb
    mov     dx,offset fcb
    mov     ah,12H
    int     21H
endm

;                                FUNCTION REQUEST 13H
delete macro fcb
    mov     dx,offset fcb
    mov     ah,13H
    int     21H
endm

;                                FUNCTION REQUEST 14H
read_seq macro fcb
    mov     dx,offset fcb
    mov     ah,14H
    int     21H
endm

```

<code>;</code>		FUNCTION REQUEST 15H
<code>write_seq</code>	<code>macro fcb</code>	
	<code>mov dx,offset fcb</code>	
	<code>mov ah,15H</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 16H
<code>create</code>	<code>macro fcb</code>	
	<code>mov dx,offset fcb</code>	
	<code>mov ah,16H</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 17H
<code>rename</code>	<code>macro fcb</code>	
	<code>mov dx,offset fcb</code>	
	<code>mov ah,17H</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 19H
<code>current_disk</code>	<code>macro</code>	
	<code>mov ah,19H</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 1AH
<code>set_dta</code>	<code>macro buffer</code>	
	<code>mov dx,offset buffer</code>	
	<code>mov ah,1AH</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 1BH
<code>def_drive_data</code>	<code>macro</code>	
	<code>mov ah,1BH</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 1CH
<code>drive_data</code>	<code>macro drive</code>	
	<code>mov dl,drive</code>	
	<code>mov ah,1CH</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 21H
<code>read_ran</code>	<code>macro fcb</code>	
	<code>mov dx,offset fcb</code>	
	<code>mov ah,21H</code>	
	<code>int 21H</code>	
	<code>endm</code>	
<code>;</code>		FUNCTION REQUEST 22H
<code>write_ran</code>	<code>macro fcb</code>	
	<code>mov dx,offset fcb</code>	
	<code>mov ah,22H</code>	
	<code>int 21H</code>	
	<code>endm</code>	

```

;                                     FUNCTION REQUEST 23H
file_size macro fcb
    mov     dx,offset fcb
    mov     ah,23H
    int     21H
endm

;                                     FUNCTION REQUEST 24H
set_relative_record macro fcb
    mov     dx,offset fcb
    mov     ah,24H
    int     21H
endm

;                                     FUNCTION REQUEST 25H
set_vector macro interrupt,handler_start
    mov     al,interrupt
    mov     dx,offset handler_start
    mov     ah,25H
    int     21H
endm

;                                     FUNCTION REQUEST 26H
create_psp macro seq_addr
    mov     dx,offset seq_addr
    mov     ah,26H
    int     21H
endm

;                                     FUNCTION REQUEST 27H
ran_block_read macro fcb,count,rec_size
    mov     dx,offset fcb
    mov     cx,count
    mov     word ptr fcb[14],rec_size
    mov     ah,27H
    int     21H
endm

;                                     FUNCTION REQUEST 28H
ran_block_write macro fcb,count,rec_size
    mov     dx,offset fcb
    mov     cx,count
    mov     word ptr fcb[14],rec_size
    mov     ah,28H
    int     21H
endm

;                                     FUNCTION REQUEST 29H
parse macro string,fc
    mov     si,offset string
    mov     di,offset fc
    push    es
    push    ds
    pop     es
    mov     al,0FH
    mov     ah,29H
    int     21H
    pop     es
endm

```



```

;                                     FUNCTION REQUEST 2AH
get_date macro
    mov     ah,2AH
    int     21H
endm

;                                     FUNCTION REQUEST 2BH
set_date macro year,month,day
    mov     cx,year
    mov     dh,month
    mov     dl,day
    mov     ah,2BH
    int     21H
endm

;                                     FUNCTION REQUEST 2CH
get_time macro
    mov     ah,2CH
    int     21H
endm

;                                     FUNCTION REQUEST 2DH
set_time macro hour,minutes,seconds
    mov     ch,hour
    mov     cl,minutes
    mov     dh,seconds
    mov     ah,2DH
    int     21H
endm

;                                     FUNCTION REQUEST 2EH
verify macro switch
    mov     al,switch
    mov     ah,2EH
    int     21H
endm

;                                     FUNCTION REQUEST 2FH
get_dta macro
    mov     ah,2FH
    int     21H
endm

;                                     FUNCTION REQUEST 30H
get_version macro
    mov     ah,30H
    int     21H
endm

;                                     FUNCTION REQUEST 31H
keep_process macro return_code,last_byte
    mov     al,return_code
    mov     dx,offset last_byte
    mov     cl,4
    shr     dx,cl
    inc     dx
    mov     ah,31H
    int     21H
endm

```

```

;                                FUNCTION REQUEST 33H
ctrl_c_ck macro action,state
    mov     al,action
    mov     dl,state
    mov     ah,33H
    int     21H
endm

;                                FUNCTION REQUEST 35H
get_vector macro interrupt
    mov     al,interrupt
    mov     ah,35H
    int     21H
endm

;                                FUNCTION REQUEST 36H
get_disk_space macro drive
    mov     dl,drive
    mov     ah,36H
    int     21H
endm

;                                FUNCTION REQUEST 38H
get_country macro country,buffer
    local   gc_01
    mov     dx,offset buffer
    mov     ax,country
    cmp     ax,0FFH
    jl      gc_01
    mov     al,0ffh
    mov     bx,country
gc_01:     mov     ah,38H
    int     21H
endm

;                                FUNCTION REQUEST 38H
set_country macro country
    local   sc_01
    mov     dx,0FFFFH
    mov     ax,country
    cmp     ax,0FFH
    jl      sc_01
    mov     al,0ffh
    mov     bx,country
sc_01:     mov     ah,38H
    int     21H
endm

;                                FUNCTION REQUEST 39H
make_dir macro path
    mov     dx,offset path
    mov     ah,39H
    int     21H
endm

;                                FUNCTION REQUEST 3AH
rem_dir macro path
    mov     dx,offset path
    mov     ah,3AH
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 3BH
change_dir macro path
    mov     dx,offset path
    mov     ah,3BH
    int     21H
endm

;                                     FUNCTION REQUEST 3CH
create_handle macro path,attrib
    mov     dx,offset path
    mov     cx,attrib
    mov     ah,3CH
    int     21H
endm

;                                     FUNCTION REQUEST 3DH
open_handle macro path,access
    mov     dx,offset path
    mov     al,access
    mov     ah,3DH
    int     21H
endm

;                                     FUNCTION REQUEST 3EH
close_handle macro handle
    mov     bx,handle
    mov     ah,3EH
    int     21H
endm

;                                     FUNCTION REQUEST 3FH
read_handle macro handle,buffer,bytes
    mov     bx,handle
    mov     dx,offset buffer
    mov     cx,bytes
    mov     ah,3FH
    int     21H
endm

;                                     FUNCTION REQUEST 40H
write_handle macro handle,buffer,bytes
    mov     bx,handle
    mov     dx,offset buffer
    mov     cx,bytes
    mov     ah,40H
    int     21H
endm

;                                     FUNCTION REQUEST 41H
delete_entry macro path
    mov     dx,offset path
    mov     ah,41H
    int     21H
endm

```



```

;                                     FUNCTION REQUEST 42H
move_ptr macro handle,high,low,method
    mov     bx,handle
    mov     cx,high
    mov     dx,low
    mov     al,method
    mov     ah,42H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 43H
change_mode macro path,action,attrib
    mov     dx,offset path
    mov     al,action
    mov     cx,attrib
    mov     ah,43H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 4400H,01H
ioctl_data macro code,handle
    mov     bx,handle
    mov     al,code
    mov     ah,44H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 4402H,03H
ioctl_char macro code,handle,buffer
    mov     bx,handle
    mov     dx,offset buffer
    mov     al,code
    mov     ah,44H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 4404H,05H
ioctl_status macro code,drive,buffer
    mov     bl,drive
    mov     dx,offset buffer
    mov     al,code
    mov     ah,44H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 4406H,07H
ioctl_block macro code,handle
    mov     bx,handle
    mov     al,code
    mov     ah,44H
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 4408H
ioctl_change macro drive
    mov     bl,drive
    mov     al,08H
    mov     ah,44H
    int     21H
endm

```

```

;
ioctl_rblock macro drive
    mov     bl,drive
    mov     al,09H
    mov     ah,44H
    int     21H
endm
FUNCTION REQUEST 4409H

;
ioctl_rhandle macro handle
    mov     bx,handle
    mov     al,0AH
    mov     ah,44H
    int     21H
endm
FUNCTION REQUEST 440AH

;
ioctl_retry macro retries,wait
    mov     bx,retries
    mov     cx,wait
    mov     al,0BH
    mov     ah,44H
    int     21H
endm
FUNCTION REQUEST 440BH

;
xdup macro handle
    mov     bx,handle
    mov     ah,45H
    int     21H
endm
FUNCTION REQUEST 45H

;
xdup2 macro handle1,handle2
    mov     bx,handle1
    mov     cx,handle2
    mov     ah,46H
    int     21H
endm
FUNCTION REQUEST 46H

;
get_dir macro drive,buffer
    mov     dl,drive
    mov     si,offset buffer
    mov     ah,47H
    int     21H
endm
FUNCTION REQUEST 47H

;
allocate_memory macro bytes
    mov     bx,bytes
    mov     cl,4
    shr     bx,cl
    inc     bx
    mov     ah,48H
    int     21H
endm
FUNCTION REQUEST 48H

```

```

;                                     FUNCTION REQUEST 49H
free_memory macro seg_addr
    mov     ax, seg_addr
    mov     es, ax
    mov     ah, 49H
    int     21H
endm

;                                     FUNCTION REQUEST 4AH
set_block macro last_byte
    mov     bx, offset last_byte
    mov     cl, 4
    shr     bx, cl
    add     bx, 17
    mov     ah, 4AH
    int     21H
    mov     ax, bx
    shl     ax, cl
    mov     sp, ax
    mov     bp, sp
endm

;                                     FUNCTION REQUEST 4B00H
exec macro path, command, parms
    mov     dx, offset path
    mov     bx, offset parms
    mov     word ptr parms[02h], offset command
    mov     word ptr parms[04h], cs
    mov     word ptr parms[06h], 5ch
    mov     word ptr parms[08h], es
    mov     word ptr parms[0ah], 6ch
    mov     word ptr parms[0ch], es
    mov     al, 0
    mov     ah, 4BH
    int     21H
endm

;                                     FUNCTION REQUEST 4B03H
exec_ovl macro path, parms, seg_addr
    mov     dx, offset path
    mov     bx, offset parms
    mov     parms, seg_addr
    mov     parms[02H], seg_addr
    mov     al, 3
    mov     ah, 4BH
    int     21H
endm

;                                     FUNCTION REQUEST 4CH
end_process macro return_code
    mov     al, return_code
    mov     ah, 4CH
    int     21H
endm

;                                     FUNCTION REQUEST 4DH
wait macro
    mov     ah, 4DH
    int     21H
endm

```



```

;                                     FUNCTION REQUEST 4EH
find_first_file macro path,attrib
    mov     dx,offset path
    mov     cx,attrib
    mov     ah,4EH
    int     21H
    endm

;                                     FUNCTION REQUEST 4FH
find_next_file macro
    mov     ah,4FH
    int     21H
    endm

;                                     FUNCTION REQUEST 54H
get_verify macro
    mov     ah,54H
    int     21H
    endm

;                                     FUNCTION REQUEST 56H
rename_file macro old_path,new_path
    mov     dx,offset old_path
    push    ds
    pop     es
    mov     di,offset new_path
    mov     ah,56H
    int     21H
    endm

;                                     FUNCTION REQUEST 57H
get_set_date_time macro handle,action,time,date
    mov     bx,handle
    mov     al,action
    mov     cx,word ptr time
    mov     dx,word ptr date
    mov     ah,57H
    int     21H
    endm

;                                     FUNCTION REQUEST 58H
alloc_strat macro code,strategy
    mov     bx,strategy
    mov     al,code
    mov     ah,58H
    int     21H
    endm

;                                     FUNCTION REQUEST 59H
get_error macro
    mov     ah,59
    int     21H
    endm

;                                     FUNCTION REQUEST 5AH
create_temp macro pathname,attrib
    mov     cx,attrib
    mov     dx,offset pathname
    mov     ah,5AH
    int     21H
    endm

```

```

;                                     FUNCTION REQUEST 5BH
create_new macro pathname,attrib
    mov     cx,attrib
    mov     dx,offset pathname
    mov     ah,5BH
    int     21H
endm

;                                     FUNCTION REQUEST 5C00H
lock macro handle,start,bytes
    mov     bx,handle
    mov     cx,word ptr start
    mov     dx,word ptr start+2
    mov     si,word ptr bytes
    mov     di,word ptr bytes+2
    mov     al,0
    mov     ah,5CH
    int     21H
endm

;                                     FUNCTION REQUEST 5C01H
unlock macro handle,start,bytes
    mov     bx,handle
    mov     cx,word ptr start
    mov     dx,word ptr start+2
    mov     si,word ptr bytes
    mov     di,word ptr bytes+2
    mov     al,1
    mov     ah,5CH
    int     21H
endm

;                                     FUNCTION REQUEST 5E00H
get_machine_name macro buffer
    mov     dx,offset buffer
    mov     al,0
    mov     ah,5EH
    int     21H
endm

;                                     FUNCTION REQUEST 5E02H
printer_setup macro index,lgth,string
    mov     bx,index
    mov     cx,lgth
    mov     dx,offset string
    mov     al,2
    mov     ah,5EH
    int     21H
endm

;                                     FUNCTION REQUEST 5F02H
get_list macro index,local,remote
    mov     bx,index
    mov     si,offset local
    mov     di,offset remote
    mov     al,2
    mov     ah,5FH
    int     21H
endm

```

```

;                                     FUNCTION REQUEST 5F03H
redir    macro    local,remote,device,value
    mov     bl,device
    mov     cx,value
    mov     si,offset local
    mov     di,offset remote
    mov     al,3
    mov     ah,5FH
    int     21H
endm

;                                     FUNCTION REQUEST 5F04H
cancel_redir macro local
    mov     si,offset local
    mov     al,4
    mov     ah,5FH
    int     21H
endm

;                                     FUNCTION REQUEST 62H
get_psp macro
    mov     ah,62H
    int     21H
endm

;
;
;*****
; General
;*****
;
display_asciiz macro asciiz_string
    local    search,found_it
    mov     bx,offset asciiz_string

;
search:
    cmp     byte ptr [bx],0
    je      found_it
    inc     bx
    jmp     short search

;
found_it:
    mov     byte ptr [bx],"$"
    display_asciiz_string
    mov     byte ptr [bx],0
    display_char 0DH
    display_char 0AH
endm

```



```

;
move_string macro source,destination,count
    push    es
    push    ds
    pop     es
    assume es:code
    mov     si,offset source
    mov     di,offset destination
    mov     cx,count
    rep movs es:destination,source
    assume es:nothing
    pop     es
endm

;
convert macro value,base,destination
    local table,start
    jmp     start
table db "0123456789ABCDEF"
;
start:
    push    ax
    push    bx
    push    dx
    mov     al,value
    xor     ah,ah
    xor     bx,bx
    div     base
    mov     bl,al
    mov     al,cs:table[bx]
    mov     destination,al
    mov     bl,ah
    mov     al,cs:table[bx]
    mov     destination[1],al
    pop     dx
    pop     bx
    pop     ax
endm

;
convert_to_binary macro string,number,value
    local ten,start,calc,mult,no_mult
    jmp     start
ten db 10
;
start:
    mov     value,0
    xor     cx,cx
    mov     cl,number
    xor     si,si

```

```

;
calc:
    xor    ax,ax
    mov    al,string[si]
    sub    al,48
    cmp    cx,2
    jl     no_mult
    push   cx
    dec    cx

;
mult:
    mul    cs:ten
    loop   mult
    pop    cx

;
no_mult:
    add    value,ax
    inc    si
    loop   calc
    endm

;
convert_date macro dir_entry
    mov    dx,word ptr dir_entry[24]
    mov    cl,5
    shr    dl,cl
    mov    dh,dir_entry[24]
    and    dh,1FH
    xor    cx,cx
    mov    cl,dir_entry[25]
    shr    cl,1
    add    cx,1980
    endm

;
pack_date macro date
    local set_bit

;
; On entry: DH=day, DL=month, CX=(year-1980)
;
    sub    cx,1980
    push   cx
    mov    date,dh
    mov    cl,5
    shl    dl,cl
    pop    cx
    jnc    set_bit
    or     cl,80h

;
set_bit:
    or     date,dl
    rol    cl,1
    mov    date[1],cl
    endm

;

```

1.13 MS-DOS システムコールにおける拡張例

```

title DISK DUMP
zero equ 0
disk_B equ 1
sectors_per_read equ 9
cr equ 13
blank equ 32
period equ 46
tilde equ 126
INCLUDE B:CALLS.EQU
;
subttl DATA SEGMENT
page +
data segment
;
input_buffer db 9 dup(512 dup(?))
output_buffer db 77 dup(" ")
db 0DH,0AH,"$"
start_prompt db "Start at sector: $"
sectors_prompt db "Number of sectors: $"
continue_prompt db "RETURN to continue $"
header db "Relative sector $"
end_string db 0DH,0AH,0AH,07H,"ALL DONE$"
;DELETE THIS
crlf db 0DH,0AH,"$"
table db "0123456789ABCDEF$"
;
ten db 10
sixteen db 16
;
start_sector dw 1
sector_num label byte
sector_number dw 0
sectors_to_dump dw sectors_per_read
sectors_read dw 0
;
buffer label byte
max_length db 0
current_length db 0
digits db 5 dup(?)
;
data ends
;
subttl STACK SEGMENT
page +
stack segment stack
dw 100 dup(?)
stack_top label word
stack ends

```



```

;
subttl MACROS
page +
;
        INCLUDE B:CALLS.MAC
;BLANK LINE
blank_line      macro      number
                  local    print_it
                  push     cx
                  call     clear_line
print_it:        mov      cx,number
                  display  output_buffer
                  loop     print_it
                  pop      cx
                  endm

;
subttl ADDRESSABILITY
page +
code          segment
start:        assume     cs:code,ds:data,ss:stack
                  mov     ax,data
                  mov     ds,ax
                  mov     ax,stack
                  mov     ss,ax
                  mov     sp,offset stack_top
;
                  jmp     main_procedure

subttl PROCEDURES
page +
;
; PROCEDURES
; READ_DISK
read_disk      proc;
                  cmp     sectors_to_dump,zero
                  jle     done
                  mov     bx,offset input_buffer
                  mov     dx,start_sector
                  mov     al,disk_b
                  mov     cx,sectors_per_read
                  cmp     cx,sectors_to_dump
                  jle     get_sector
                  mov     cx,sectors_to_dump
get_sector:    push     cx
                  int     disk_read
                  popf
                  pop     cx
                  sub     sectors_to_dump,cx
                  add     start_sector,cx
                  mov     sectors_read,cx
                  xor     si,si
done:          ret
read_disk      endp

```

```

;
;CLEAR_LINE
clear_line      proc;
                 push    cx
                 mov     cx,77
                 xor     bx,bx
move_blank:     mov     output_buffer[bx], ' '
                 inc     bx
                 loop    move_blank
                 pop     cx
                 ret
clear_line      endp
;
;PUT_BLANK
put_blank       proc;
                 mov     output_buffer[di], " "
                 inc     di
                 ret
put_blank       endp
;
;
setup          proc;
display        start_prompt
get_string     4,buffer
display        crlf
convert_to_binary digits,
current_length,start_sector
mov            ax,start_sector
mov            sector_number,ax
display        sectors_prompt
get_string     4,buffer
convert_to_binary digits,
current_length,sectors_to_dump
ret
setup          endp
;
;CONVERT_LINE
convert_line    proc;
                 push    cx
                 mov     di,9
                 mov     cx,16
convert_it:     convert  input_buffer[si],sixteen,
output_buffer[di]
                 inc     si
                 add     di,2
                 call    put_blank
                 loop    convert_it
                 sub     si,16
                 mov     cx,16
                 add     di,4
display_ascii: mov     output_buffer[di],period
                 cmp     input_buffer[si],blank
                 jl      non_printable
                 cmp     input_buffer[si],tilde
                 jg      non_printable

```

```

printable:      mov     dl, input_buffer[si]
                mov     output_buffer[di], dl
non_printable:  inc     si
                inc     di
                loop    display_ascii
                pop     cx
                ret
convert_line    endp
;
;DISPLAY_SCREEN
display_screen  proc;
                push    cx
                call    clear_line
;
                mov     cx, 17
;I WANT length header
                dec     cx
;minus 1 in cx
                xor     di, di
move_header:    mov     al, header[di]
                mov     output_buffer[di], al
                inc     di
                loop    move_header    ;FIX THIS!
;
                convert sector_num[1], sixteen,
                output_buffer[di]
                add     di, 2
                convert sector_num, sixteen,
                output_buffer[di]
                display output_buffer
                blank_line 2
dump_it:        mov     cx, 16
                call    clear_line
                call    convert_line
                display output_buffer
                loop    dump_it
                blank_line 3
                display continue_prompt
                get_char_no_echo
                display crlf
                pop     cx
                ret
display_screen endp
;
;
;  END PROCEDURES
subttl MAIN PROCEDURE
page +
main_procedure: call    setup
check_done:      cmp     sectors_to_dump, zero
                jng     all_done
                call    read_disk
                mov     cx, sectors_read

```



```
display_it:      call    display_screen
                  call    display_screen
                  inc     sector_number
                  loop    display_it
                  jmp     check_done
all_done:        display end_string
                  get_char_no_echo
code              ends
                  end     start
```


第2章

MS-DOSデバイスドライバ

2.1 デバイスドライバとは

IO.SYS は、標準で組み込まれているデバイスドライバによって構成されていると言えます。IO.SYS は、MS-DOS の BIOS を形成し、MS-DOS からコールされます。アプリケーションプログラムによる I/O リクエストは、MS-DOS によって IO.SYS を通じて処理されます。

MS-DOS では、BIOS などを書き直すことなく、新しいデバイス(例：プリンタ、プロッタ、またはマウス入力デバイスなど)を追加することができます。つまり、MS-DOS の BIOS はユーザー側で“構成可能”であり、既存のドライバの上に新しいドライバを追加できるのです。ユーザーは、システムの起動（ブート）時に参照される CONFIG.SYS ファイル内の“DEVICE=”コマンドを用いて、デバイスドライバを登録することにより、新規のデバイスドライバ(以下、単にデバイスドライバと呼びます)を容易に追加できます。第2章では、IO.SYS ファイル内のドライバ（標準で組み込まれているデバイスドライバ）を追加されるデバイスドライバと区別するために標準のデバイスドライバと呼びます。

起動時には、少なくとも5個の標準のデバイスドライバが存在していなければなりません。これらのドライバは、リンクされたリストで示されます。各ドライバの“ヘッダ”には、つぎのドライバに対する DWORD ポインタが含まれます。チェーン内の最後のドライバには、-1, -1（全ビット共オン）というエンドオブリストマーカがあります。

チェーン内の各ドライバには、ストラテジ、および割り込みという2個のエントリポイントがあります。2個のエントリポイントは、将来の MS-DOS のバージョンでマルチタスクをサポートするときのために設けられています。

マルチタスク環境では、I/O を非同期で行わなければなりません。これを実現するために本来は、以下のような手順で処理が行われる必要があります。

- ・ストラテジルーチンは呼び出されたらリクエストを内部的に待ち行列（キュー）に登録し、即座にリターンします。
- ・割り込みルーチンは割り込みにより起動され、内部的な待ち行列からリクエストを取り出し、それを処理します。
- ・リクエストが完了すると、割り込みルーチンは実行済みフラグをセットします。MS-DOS は、定期的にこの実行済みフラグをチェックし、実行済みフラグのセットされているリクエストを探し、そのリクエストの終了を待っているプロセスに制御を移します。

このようにリクエストを待ち行列に入れる方法をとった場合、一時に複数のリクエストが待ち状態になる可能性があるため、レジスタによってI/O 情報を受け渡すことは不可能になります。そのため、MS-DOS のデバイスインターフェイスは“パケット”を用いてリクエスト情報を渡します。このリクエストパケットはサイズ、フォーマットが可変であり、つぎの2つの部分から構成されます。

1. 静的リクエストヘッダセクション。すべてのリクエストで同じフォーマットを持ちます。
2. リクエストのタイプごとに固有の情報を持つセクション。

ドライバが呼び出されるときは、パケットに対するポインタが渡されます。将来のMS-DOS のマルチタスクバージョンでは、このパケットはMS-DOS によって保護される、すべてのI/O 待ち行列のグローバルチェーンにリンクされます。

現在のMS-DOS のバージョンでは、I/O は非同期では行われなため、グローバルまたはローカルの待ち行列を提供していません。一時的に待ち状態となりうるのは1個のリクエストだけなので、2個のエントリポイントは本来の目的では使用されず、処理は以下のようになります。

- ・ストラテジルーチンは特定の場所にパケットのアドレスを格納し、MS-DOS にリターンします。
- ・割り込みルーチンはストラテジルーチンの直後に呼び出され、パケットをもとにリクエストを処理します。

割り込みルーチンから戻った時点でリクエストは完了したものと見なされます。

インストール可能なデバイスドライバは、ファイルの先頭にデバイスヘッダを持つ、BIN 形式(.BIN;コアイメージ)または、EXE 形式(.EXE) フォーマットでファイルを作成する必要があります。デバイスヘッダのリンクフィールドは、-1, -1 に初期設定されていなければなりません(SYSINIT がこれをセットします)。

複数のデバイスドライバを1つのファイルに納める場合、各デバイスドライバのデバイスヘッダにはリスト内の次のデバイスを指すポインタをセットします。最後のデバイスヘッダは-1, -1 で初期設定されていなければなりません。

MS-DOS のバージョン3.0以降ではデバイスドライバは、BIN 形式、EXE 形式のどちらのフォーマットでも使用することができます。ただし、バージョン3.0以前のMS-DOS ではBIN 形式のデバイスドライバしか使用できないため、どちらのバージョンでも動作可能なデバイスドライバを作成する場合、BIN 形式とする必要があります。

2.2 デバイスドライバのフォーマット

デバイスドライバは、MS-DOS に新しいシステムハードウェアを管理するI/O 環境を提供するためのプログラムセグメントです。デバイスドライバの先頭には、特殊なヘッダがあり、

このヘッダでは、デバイスドライバであることを識別し、エントリポイントを定義し、そしてデバイスの各種のアトリビュート（属性）を記述します。

OPEN/CLOSE/RM 機能（オープン/クローズ/リムーブ）をサポートしている場合、ビット 11 を 1 にします。

注意：

デバイスドライバには、ファイルに ORG 100H (COM ファイルのような) を使用しないでください。デバイスドライバはプログラムセグメントプレフィクスを使用せずに、単にロードされるだけです。したがって、このファイルの起点は 0 でなければなりません (ORG 0 または ORG ステートメントなし)。

デバイスドライバには、つぎの 2 種類があります。

- 1 キャラクタデバイスドライバ
- 2 ブロックデバイスドライバ

キャラクタデバイスとは、コンソール、プリンタ、RS-232C コミュニケーションポートのようにシリアルな（一連の）文字の I/O を行うもので、これらの装置には特別の名前（ファイル名）が付けられています (CON, PRN, AUX など)。ユーザーは、これらの装置の名前を指定することで I/O を扱うためのチャネル（ハンドルまたは FCB）をオープンすることができます。

ブロックデバイスは、システムの“ディスクドライブ”のことで、ブロック単位（通常、物理セクタサイズ単位）でランダムな I/O（入出力）を扱うことができます。ブロックデバイスにはファイル名は付けられないので、直接にオープンすることができません。ブロックデバイスはドライブ名 (A:, B:, C:) によって識別されます。

ブロックデバイスは、ユニットで構成されています。1つのドライバが、複数のディスクドライブを処理することができます。たとえば、ALPHA というデバイスドライバは、A:, B:, C:, および D: というドライブを処理できるとします。これは、4つのユニット (0 ~ 3) を定義しており、そのための4つのドライブ文字を使用するという意味です。ドライバリスト中のドライブの順番によって、どのユニットがどのドライブ文字と対応するかが決定されます。ALPHA というドライバがデバイスリスト中の先頭にあるブロックドライバで4つのユニット (0 ~ 3) を定義している場合、ドライブは A:, B:, C:, D: に、BETA が2番目にあるブロックドライバで3つのユニット (0 ~ 2) を定義している場合、E:, F:, G: になります。

ブロックデバイスユニットは、理論上は 63 まで使用できますが、デバイスインストールコードで、ドライブを表す文字が "Z" (5AH) より大きくなる場合には、そのデバイスの登録は不可能です。標準の BIOS 内に存在するブロックデバイスドライバ（システムのディスクドライブ）はすべて、ブロックデバイスドライバより先に置かれます。

注意：

キャラクタデバイスには 1 つしか名前が付いていないので、複数のユニットを定義することはできません。

2.3 デバイスドライバの作成方法

MS-DOS でデバイスドライバを作成するために、ファイルの開始点にデバイスヘッダが付けられているバイナリファイル（COM 形式、または EXE 形式のファイル）を作成します。デバイスドライバの場合、コードの起点は 100H ではなく 0H でなければなりません。リンクフィールド（つぎのデバイスヘッダに対するポインタ）は、このファイル内にデバイスドライバが 1 つしかない場合、-1、-1 でなければなりません。アトリビュートフィールドおよびエントリポイントは、正しくセットしなければなりません。

キャラクタデバイスの場合、名前フィールドにデバイス名をセットします。この名前には、すべての文字をファイル名として 8 文字使用することができます。

この名前が 8 文字未満の場合は、スペース (20H) を詰めることによって 8 文字にしなければなりません。デバイス名には、コロン (:) は含まれません。"CON" は "CON:" と同じものですが、これは、デフォルトの MS-DOS コマンドインタプリタ (COMMAND.COM) の特性であって、デバイスドライバまたは MS-DOS の特性ではありません。キャラクタデバイス名はすべて、この方法で取り扱われます。

MS-DOS では、標準のデバイスを使用する前に常にデバイスドライバを使用するので、新規の CON（コンソール）デバイスを登録する場合、名前を単に "CON" のみに指定してください。新規の CON デバイスのアトリビュート内に標準入出力デバイスビットをセットすることを忘れないでください。最初に一致するものが現われた時点でデバイスリストの走査が停止するので、このデバイスドライバが優先します。

BIOS で他のデバイスドライバを置き換えるのと同じ方法で、標準のディスクブロックデバイスドライバと、デバイスドライバを置き換えるのは、不可能です。ブロックデバイスドライバは、IO.SYS のデフォルトのディスクドライバによって直接サポートされないデバイスに関してのみ、使用することができます。

注意：

MS-DOS では、メモリの任意の位置にドライバを登録できるので、アドレスの離れたメモリを参照する場合、注意が必要です。ユーザーのドライバがロードされる場合、常に同じ位置にロードされるとは限りません。

2.3.1 デバイスストラテジルーチン

このルーチンは、デバイスドライバのサービスリクエストが発生するたびに MS-DOS によって呼び出され、これらのリクエストを、デバイス割り込みルーチンによって処理される順序で待ち行列に入れます。

このような待ち行列の機能は、非同期 I/O がサポートされるマルチタスク実行時の環境において、非常に重要な機能です。V 3.3 では、この種の機能をサポートしていないので一時にサービスできるのは、1つのリクエストだけです。

通常、このルーチンは非常に短く、2.12 のコーディング例では、おののおのリクエストを単に、単一のポインタ領域に格納しています。

2.3.2 デバイス割り込みルーチン

このルーチンには、リクエストをサービスするためのコードのすべてが含まれます。このルーチンは実質的にハードウェアとインターフェイス (ROM の BIOS コールなど) を取ります。通常このルーチンは、サポートされる特定のコマンドコードを処理するための一連のプロシージャの他、ある種の "EXIT" およびエラーハンドリングルーチンによって構成されています。2.12 のコーディング例を参照してください。

2.4 デバイスドライバの登録

V 2.0 以降の MS-DOS では、新規のデバイスドライバを起動時に自由に登録することが可能で、これは CONFIG.SYS ファイルを読み込むことにより IO.SYS 内の INIT コードによって実行されます。

MS-DOS は、つぎの方法によってデバイスドライバのコールを行います。

1. ストラテジエントリに対して FAR コールを行う。
2. リクエストヘッダ中のデバイスドライバの情報をストラテジルーチンに渡す。
3. 割り込みエントリに対して FAR コールを行う。

この方法により、将来の MS-DOS のバージョンでマルチタスク処理をサポートする際に、容易に対応できるようになっています。

2.5 デバイスヘッダ

デバイスドライバの開始点にはデバイスヘッダを付ける必要があります。このヘッダは、つぎのような内容のものです。

2 ワード	つぎのデバイスへのポインタ(ファイルの中で最後または唯一のドライバの場合-1 にセットしなければいけません)
1 ワード	アトリビュート (属性) ビット15 = $\begin{cases} 0 & \text{ブロックデバイス} \\ 1 & \text{キャラクタデバイス} \end{cases}$ ビット15 = 1 の場合さらに ビット 0 = 1 カレント標準入力デバイス 1 = 1 カレント標準出力デバイス 2 = 1 カレント NUL デバイス 3 = 1 カレント CLOCK デバイス 4 = 1 特殊な装置 5 予約域 0 であること 7 ~ 10 予約域 0 であること ビット14 IOCTL ビット ビット13 キャラクタデバイス (ビット15 = 1) の場合 OUTPUT UNTIL BUSY ブロックデバイス (ビット15 = 0) の場合 NON FAT ID ビット12 予約域 0 であること ビット11 OPEN/CLOSE/RM サポート ビット 6 V3.3 ビット
1 ワード	デバイスストラテジ・エントリポインタのポインタ
1 ワード	デバイス割り込みエントリポインタのポインタ
8 バイト	キャラクタデバイスファイル名 キャラクタデバイスはデバイス名でセットします。 ブロックデバイスの場合、先頭バイトはユニット数を表します。

デバイスエントリポインタは、ワードを使用します。これらのエントリポインタは、このテーブルをポイントするために同じセグメント番号からのオフセットでなければなりません。たとえば、XXX:YYY がこのテーブルの開始点をポイントとしている場合、XXX:ストラテジと XXX:割り込みがエントリポイントになります。

2.5.1 つぎのデバイスヘッダフィールドに対するポインタ

つぎのデバイスヘッダフィールドに対するポインタは2ワードのフィールド（オフセット、セグメントの順）で、デバイスドライバのロード時に、システムリスト中のつぎのドライバをポイントするようにMS-DOSによってセットされます。ファイル内に1つしかデバイスドライバが存在しない場合は、ロードされる前に（ファイルとしてディスク上にあるとき）このフィールドを-1にセットしなければなりません。ファイル内に複数のドライバが存在する場合、2ワードポインタの先頭のワードは、つぎのドライバのデバイスヘッダのオフセットでなければなりません。

注意：

COM形式(.COM)ファイル内に複数のデバイスドライバが存在する場合、このファイル内の最終ドライバのこのフィールドは-1にセットされていなければなりません。

2.5.2 アトリビュート(属性)フィールド

アトリビュート(属性)フィールドは、このドライバが扱うデバイスのタイプを識別します。これらのビットは、ブロックとキャラクタデバイスを区別するほか、選択されたキャラクタデバイスに対し、特殊な取り扱いを行います（アトリビュートワード内のあるビットをあるデバイスのタイプに関してのみ定義する場合は、その他のデバイスのタイプ用のドライバでは、そのビットを0にしなければなりません）。

たとえばユーザーが、標準入出力用にしたい新規のデバイスドライバを所有しているとします。このドライバを登録するとともに、現在の標準入出力（CON：コンソール）を無効にすることをMS-DOSに通知しなければなりません。この作業は、アトリビュート(属性)をセットすることによって行います。そのためには、0ビット目および1ビット目を1にセットします（これらのビットは、別々の役割を果たします）。同様に新規のクロックデバイスは、この該当するビットをセットすることによって登録することができます（詳細については、“2.10 クロックデバイス”を参照してください）。NULデバイスのアトリビュートは存在しますが、この装置に対して再割り当てを行うことはできません。このアトリビュートで、NULデバイスを使用中かどうかをMS-DOSが調べることができます。

ブロックデバイス用のNON FAT IDビットは、BUILD BPB（BIOSパラメータブロック）デバイスコールの動作に影響を与えます。また、このNON FAT IDビットは、キャラクタデバイスでは、意味が異なります。このビットは、そのデバイスがOUTPUT UNTIL BUSYデバイスコールを実施することを表します。

IOCTLビットは、キャラクタデバイスにもブロックデバイスに対しても意味があります。

IOCTL ファンクションを使用すれば、デバイスドライバは、デバイスドライバ自身の目的のために（たとえば、ボーレート、ストップビットなどをセットするために）データの転送および取得を行うことができます。渡された情報の処理方法はデバイスによって異なりますが、通常の I/O 要求として処理してはいけません。このビットは、ファンクションリクエスト 44H の IOCTL システムコールで、デバイスドライバがコントロールストリングを処理可能かどうかを MS-DOS に通知するものです。

ドライバがコントロールストリングを処理できない場合、最初にこのビットを 0 にセットしなければなりません。これにより、デバイスとの間でコントロールストリングの転送または取得が行われようとした場合（ファンクション 44 H によって）、MS-DOS はエラーを返します。コントロールストリングを処理可能なデバイスの場合、IOCTL ビットを 1 にセットします。この種類のデバイスの場合、IOCTL ストリングを転送および取得するために、MS-DOS は IOCTL 入出力デバイスファンクションコールを行います。

OPEN/CLOSE/RM ビットは、V 3.1 以降の MS-DOS に対し、このドライバが V 3.1 以降の追加機能をサポートするかどうかを通知します。

V 3.0 以前で作成したドライバをサポートするためには、そのことを検出する必要があります。このビットは、V 3.0 以前で予約されており、0 です。新しいデバイスはすべて、OPEN、CLOSE、および REMOVABLE MEDIA コールをサポートしなければならず、このビットを 1 にセットしなければなりません。

V 3.0 以前では、これらのコールを行わないので、V 3.0 以前のドライバは V 3.1 以降でも互換性が保たれます。

V3.3 ビットは、V3.3 以降の MS-DOS に対し、このドライバがファンクション 440EH (Get Logical Drive Map) と 440FH (Set Logical Drive Map) による論理ドライブのマップをサポートするかどうかを通知します。

このビットはさらに、ファンクション 440CH (Generic IOCTL for handles) とファンクション 440DH (Gereric IOCTL for Block devices) のサポートも意味します。

2.5.3 ストラテジと割り込みルーチン

これらの 2 つのフィールドは、ストラテジおよび割り込みルーチンのエントリポイントへのポインタです。これらは 1 ワードの値を持っているので、デバイスヘッダの同じセグメント内に存在しなければなりません。

2.5.4 名前フィールド

これは8バイトのフィールドで、キャラクタデバイスの名前またはブロックデバイスのユニット数が入っています。ブロックデバイスの場合は、ユニット数を先頭のバイトに入れます。MS-DOSでは、このロケーションにドライバのINITコードによって返された値が入れられるので、このフィールドは選択できます。詳細については、2.4の「デバイスドライバの登録」を参照してください。

2.6 リクエストヘッダ

MS-DOSはデバイスドライバのコールを行う場合、ES:BX内のリクエストヘッダをストラテジエントリポイントに渡します。これは固定長ヘッダで、処理に必要なデータが入っています。マシンステートを保存する（たとえば、制御が渡されるときすべてのレジスタを保存し、抜け出すときこれらレジスタを復元する）のは、デバイスドライバの役目です。ストラテジまたは割り込みコールが行われるとき、スタック内には約20個のデータを入れるのに十分な領域がありますが、これ以上のスタックを必要とする場合ドライバによって必要なスタックをセットします。

つぎの図で、リクエストヘッダの説明を行います。

1 バイト	レコードの長さ このリクエストヘッダをバイト単位で表した長さ
1 バイト	ユニットコード 処理に用いるサブユニット（キャラクタデバイスの場合は意味を持ちません）
1 バイト	コマンドコード
1 ワード	ステータス
8 バイト	予約域 2つのDWORDのリンクのための予約域、1つはMS-DOSキューのため、もう1つはデバイスキューのリンク用

つぎにリクエストヘッダフィールドについて解説します。

2.6.1 レコード長

このフィールドはリクエストヘッダの大きさ（バイト単位）です。

2.6.2 ユニットコード

ユニットコードフィールドは、ユーザーのデバイスドライバ内のどのユニットに対してリクエストが行われるかを、識別するものです。たとえば、ユーザーのデバイスドライバで3つのユニットが定義されている場合、ユニットコードフィールドの値は、0, 1, 2 になる可能性があります。

2.6.3 コマンドコードフィールド

リクエストヘッダ内のコマンドフィールドには、つぎの値を入れることができます。

コマンドコード	ファンクション
0	INIT
1	MEDIA CHECK (ブロックデバイスドライバのみ)
2	BUILD BPB (ブロックデバイスドライバのみ)
3	IOCTL INPUT (デバイスドライバが IOCTL 機能を持つ場合のみ)
4	INPUT (リード)
5	NON-DESTRUCTIVE INPUT NO WAIT (キャラクタデバイスのみ)
6	INPUT STATUS (キャラクタデバイスのみ)
7	INPUT FLUSH (キャラクタデバイスのみ)
8	OUTPUT (ライト)
9	OUTPUT (ライトとペリファイが行われます)
10	OUTPUT STATUS (キャラクタデバイスのみ)
11	OUTPUT FLUSH (キャラクタデバイスのみ)
12	IOCTL OUTPUT (デバイスドライバが IOCTL 機能を持つ場合のみ)
13	DEVICE OPEN (デバイスドライバが OPEN/CLOSE/RM 機能を持つ場合のみ)
14	DEVICE CLOSE (デバイスドライバが OPEN/CLOSE/RM 機能を持つ場合のみ)
15	REMOVABLE MEDIA (OPEN/CLOSE/RM 機能を持つブロックデバイスの場合のみ)
16	OUTPUT UNTIL BUSY (キャラクタデバイスにおいて、ビット 13 をセットしている場合のみ)
19	Generic IOCTL (V3.3 ビット (ビット 6) が 1 のデバイスのみ)
20	DEINSTALL (キャラクタデバイスのみ)
23	Get Drive Map (V3.3 ビット (ビット 6) が 1 のブロックデバイスのみ)
24	Set Drive Map (V3.3 ビット (ビット 6) が 1 のブロックデバイスのみ)

2.6.4 ステータスフィールド

リクエストヘッダのステータスフィールドは、つぎの図で示すような内容です。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
E R R	予 備					B U S	D O N	エラー・コード(ビット15がオン)							

ステータスフィールドは、ドライバ割り込みルーチンに制御が渡されるときはゼロで、ルーチンから戻るときにセットされます。

8ビット目は DONE (処理済) ビットで、セットされた場合は、動作が完了したことを意味します。ドライバから抜け出すときに 1 にセットされます。

15ビット目は、エラービットです。セットされた場合、下位 8 ビットがエラーを示します。エラーの意味は、つぎのとおりです。

00H	ライトプロテクト(保護)違反
01H	無効なユニット
02H	ドライブの準備ができていない
03H	無効なコマンド
04H	CRC エラー
05H	不正なドライブリクエストの長さ
06H	シークエラー
07H	無効なメディア
08H	セクタが存在しない
09H	プリンタの用紙切れ
0AH	ライトエラー
0BH	リードエラー
0CH	一般的なエラー
0DH	予備
0EH	予備
0FH	不正なディスクの交換

9ビット目は BUSY ビットで、STATUS コールおよび REMOVABLE MEDIA コールによってのみセットされます。

2.7 デバイスドライバファンクション

デバイスドライバは9個のファンクション（1つ以上のファンクションの組み合わせ）から構成されます。さらにそのファンクションが種々のコマンドラインで構成される場合があります。

すべてのストラテジルーチンは、リクエストヘッダをポイントしている ES:BX を使用してコールします。割り込みルーチンは、リクエストヘッダのポインタをストラテジルーチンが持つキューから得ます。リクエストヘッダのコマンドコードは、行うべきファンクションとリクエストヘッダに続くデータをドライバに通知します。

注意：

すべての2ワードポインタは、最初にオフセット、つぎにセグメントが記憶されています。

2.7.1 INIT

コマンドコード=0

INIT ES:BX ←

13 バイト	リクエストヘッダ
1 バイト	ユニット数
2 ワード	エンドアドレス
2 ワード	BPB 配列に対するポインタ (キャラクタデバイスはセットしない。)
1 バイト	ブロックデバイス番号

各々のデバイスドライバに関して定義されるファンクションの1つに、INIT があります。このルーチンは、デバイスが登録される時に、1回だけ呼び出されます。INIT ルーチンは、エンドアドレス（デバイスドライバの常駐の部分の終りに対する DWORD ポインタ）を返さなければなりません。これは、1回しか必要のない初期設定コードを削除し、スペースを節約する目的で使うことができます。

ドライバは、ユニット数、エンドアドレス、および BPB 配列に対するポインタをセットします。しかし、デバイスドライバに入る時点で、(ブロックデバイスにおける)BPB 配列に対してドライバによってセットされる DWORD は、このドライバがロードされる原因となった、

CONFIG.SYS 内の行の "device =" の後のキャラクタを指しています。そのためドライバは、CONFIG.SYS の起動行を調べて、ドライバに渡すべきパラメータを見つけ出すことができます。この行はリターンまたはラインフィードで終わります。このデータは読み出し専用であり、これによってデバイスは CONFIG.SYS の行を調べて引数を見つけ出すことができます。

```
device=%dev%vt 52. sys/1
```

↑

BPB アドレスは、ここを示します。

さらに、ブロックデバイスに関する場合ですが、このドライバによって定義された最初のユニット(A=0)に割り当てられるドライブ番号が、ブロックデバイス番号フィールドにセットされます。これもやはり読み出し専用です。

キャラクタデバイスについては、エンドアドレスパラメータが返されなければなりません。これはドライバより上の最初の使用可能なバイトに対するポインタであり、初期設定のコードを捨てるために使用することができます。

ブロックデバイスは、つぎの情報を返さなければなりません：

1. ユニット数を返す必要があります。MS-DOS はこれを用いて、論理デバイス名を決定します。登録コールの時点で、現在の最大の論理デバイス文字が F であり、INIT ルーチンがユニット数として 4 を返す場合、これらのデバイスの論理名は、G、H、I および J です。このマッピングは、デバイスリストにおけるドライバの位置と、デバイス上のユニット数（デバイス名フィールドの最初のバイトに格納されている）によって決定されます。
2. BPB(BIOS パラメータブロック)へのワードオフセット(ポインタ)の配列に対する DWORD ポインタを返す必要があります。MS-DOS は、デバイスドライバによって渡された BPB を用いて、内部構造を作成します。デバイスドライバによって定義される各々のユニットに関して、この配列の中に 1 個ずつのエントリが必要です。この方法を用いることにより、もしすべてのユニットが同じであれば、すべてのポインタが同一の BPB を指すことになり、スペースが節約できます。デバイスドライバが 2 個のユニットを定義する場合は、DWORD ポインタは、2 個の 1 ワードオフセットの最初の方を指し、これらのオフセットがさらに BPB を指します。BPB のフォーマットについては 2.7.3 "BUILD BPB" を参照してください。

DOS 内部構造は、フリーポインタによって指されるバイトを起点として構成されるので、このワードオフセットの配列は(リターンによってセットされるフリーポインタの下側を)保護されなければなりません。定義するセクタサイズは、初期設定中に標準のデバイスドライバ (BIOS) によってセットされる最大セクタサイズ以下でなければなりません。これ

が異なる場合は、初期設定は失敗します。

3. ブロックデバイスの INIT で返すべき最後のデータは、メディアディスクリプタバイトです。このバイトは、MS-DOS 自身に対しては何の意味も持ちませんが、MS-DOS が特定のドライバユニットに関して現在どのようなパラメータを使用しているかが分かるよう、デバイスに渡されます。

ブロックデバイスには、“ダム” または “スマート” の2種類があります。

ダムデバイスは、種々の可能なメディアとドライブの組合せについて、ユニット (同時に DOS 内部構造) を定義します。たとえば、ユニット 0=ドライブ 0 片面、ユニット 1=ドライブ 0 両面、などです。この場合、メディアディスクリプタバイトは、何の意味も持ちません。

一方、スマートデバイスは、1つのユニットについて、複数のメディアの使用を認めます。この場合、INIT で返される BPB テーブルでは、サポートされる最大のメディアを収容できる、十分なスペースが定義されていなければなりません。スマートドライバは、メディアディスクリプタバイトを用いて、ユニット内に現在あるメディアに関する情報を渡します。

メディアディスクリプタバイトに関する詳細は、2.8 の “メディアディスクリプタバイト” を参照してください。

キャラクタ系デバイスドライバを、ADDDRV、DELDRV コマンドで動的に登録、削除可能にするには、エンドアドレスに加えて次の情報を返す必要があります。

デバイスドライバは、ドライバに入った時点でセットされている「BPB 配列に対するポインタ」のすべてのビットを反転してセットしなおすことにより、ADDDRV コマンドに対して動的な登録、削除機能をサポートしているドライバであることを宣言します。

ドライバに入った時点の

BPB 配列に対するポインタ → 12345678H

↓ すべてのビットを反転

ドライバから返される

BPB 配列に対するポインタ → EDCBA987H

このことにより、DELDRV コマンドにてデバイスドライバを削除する場合に後述のファンクション 20(DEINSTALL) がコールされるようになります。

キャラクタ系デバイスドライバを、動的に登録、削除可能とする場合、以下の点に注意してください。

1. そのデバイスドライバが、ハードウェアのモードなどシステムの情報を変更する可能性がある場合、DEINSTALL ファンクション(ファンクション 20)で元に戻さなければなりません。そのため、この INIT ファンクションでは、変更する可能性のある情報を自身のメモリ内にセーブしておく必要があります。
2. ADDDRV コマンドは、割り込みベクタをすべて専用の領域にセーブしておきます。この内容は、DELDV コマンドによって復旧されるため、デバイスドライバは割り込みベクタの内容をセーブ、リストアする必要はありません。

注意：

INIT ファンクション処理中には、MS-DOS のファンクションリクエスト(INT 21H)のファンクション 01H~0CH, 25H, 30H, 35H のみ使用できます。

2.7.2 MEDIA CHECK

コマンドコード=1

MEDIA CHECK ES:BX →

13バイト	リクエストヘッダ
1バイト	BPB からのメディアディスクリプタ
1バイト	返された値
2ワード	デバイスアトリビュートフィールドのビット 11 がセットされ、かつ上段の 1 バイトフィールドに "1" が返されたときの、以前のボリューム ID へのポインタ

MEDIA CHECK ファンクションは、ブロックデバイスでのみ使用されます。このファンクションは、ファイルのリードまたはライト以外の、ペンディング中のドライブアクセスコール（例：オープン、クローズ、削除、およびリネームなど）がある時に呼び出されます。その目的は、ドライブ内のメディアが変更されているかどうかを判定することです。ドライバが、メディアは変更されていないことを確認できたら（ドアロックまたは他のインターロックメカニズムによる）、MS-DOS は FAT を再び読み込んだり、ディレクトリアクセスのたびにインメモリバッファを無効にする必要がないので、MS-DOS の処理効率は高められます。

MS-DOS に対するこのようなディスクアクセスコール（ファイルのリードまたはライト以外）が発生すると、つぎのような一連の事象が発生します。

1. MS-DOS は、ドライブ文字（ドライブ名のコロンなし）を、特定のブロックデバイスのユニット番号に変換します。
2. デバイスドライバが呼び出され、ディスクが変更されているかどうかを調べるために、そのサブユニットに関するメディアチェックをリクエストします。MS-DOS は、以前のメディアディスクリプタバイトを渡します。ドライバによって返される値は、つぎのとおりです。

メディアが変更されていない場合.....	1
変更されたかどうか不明な場合.....	0
メディアが変更された場合.....	-1
エラー.....	それ以外の値

メディアが変更されていない場合、MS-DOS は続いてディスクアクセスを行います。

返された値が“不明”の場合は、ディスクセクタのうち、修正されているけれども、このユニットに関してまだディスクに書き戻されていないものがあれば、MS-DOS はディスクは変更されていないものと見なし、処理を続けます。MS-DOS はこのユニットに関する他のバッファを無効にし、BUILD BPB デバイスコールを行います（つぎの3を参照）。

メディアが変更されている場合は、MS-DOS は、書き込みを待っている修正済みデータのあるバッファも含めて、このユニットに関連するすべてのバッファを無効にし、BUILD BPB コールを用いて、新しい BIOS パラメータブロックをリクエストします（つぎの3を参照）。

3. BPB が返されると、MS-DOS は、新しい BPB からドライブに関する内部構造を修正し、ディレクトリおよび FAT を読み込んだ後、続けてアクセスを行います。

以前のメディアディスクリプタバイトが、デバイスドライバに渡される点に注意してください。以前のメディアディスクリプタバイトが新しいものと同じである場合には、ディスクが変更され、新しいディスクがドライブ内にある可能性があります。したがって、そのユニットに関する FAT、ディレクトリおよびメモリ内にバッファされたデータセクタはすべて、無効であると考えられます。

ドライバのデバイスアトリビュートワードのビット 11 が 1 の場合、ドライバが -1 (メディアは変更された) を返した場合は、ドライバは DWORD ポインタを、以前のボリューム ID フィールドにセットしなければなりません。DOS が、“メディアは変更された”とし、DOS バッファキャッシュの状態に基づくエラーであると判定した場合は、DOS はデバイスのために 0FH エラーを発生させます。ドライバがボリューム ID サポートを実施していないけれども、ビット 11 をセットしている場合は、ストリング “NO NAME” に対する静的ポインタを 0 にセットしなければなりません。

ドアロックがない問題に対する解決方法を、つぎに示します。

ユーザーがディスクを短い一定の時間 (ハードウェアに依存します:たとえば 2 秒) 以内で変更するのは普通は不可能でしょう。そのため、ディスクアクセスの短い一定の時間以内に MEDIA CHECK が発生した場合は、ドライバは “1”, すなわち “メディアは変更されていない” を報告します。これにより、処理効率が著しく改善されます。

注意:

返された BPB 中のメディアディスクリプタバイトが、以前のメディアディスクリプタバイトと同じである場合には、MS-DOS はディスクのフォーマットが同じであると見なし (ディスクが変更されているとしても)、ディスクの内部構造を更新するステップを飛ばします。したがって、BPB はすべて、FAT ID バイトとは無関係に、ユニークなメディアバイトを持っていなければなりません。

2.7.3 BUILD BPB (BIOS パラメータブロックの作成)

コマンドコード=2

BUILD BPB ES:BX →

13バイト	リクエストヘッダ
1バイト	BPB からのメディアディスクリプタ
2ワード	転送アドレス (デバイスアトリビュートフィールドのビット 13 に依存する, 1 セクタ分のスクラッチスペースまたは FAT の最初のセクタへのポインタ)
2ワード	BPB に対するポインタ

BUILD BPB ファンクションは、ブロックデバイスについてのみ使用します。MEDIA CHECK ファンクションの項で述べたように、BUILD BPB ファンクションは、先行する MEDIA CHECK コールにより、ディスクが変更されているか、変更された可能性があること

を示された場合に、任意の時点で呼び出されます。デバイスドライバは、BPB に対するポインタを返さなければなりません。この点は、BPB へのワードオフセットの配列に対するポインタが返される INIT コールとは異なります。

BUILD BPB コールは、1 セクタバッファに対する DWORD ポインタ（転送アドレス）を入手します。このバッファの内容は、アトリビュート（属性）フィールド内の NON FAT ID ビット（ビット 13）によって決定されます。

このビットが 0 の場合は、このバッファには、最初の FAT の最初のセクタが含まれます。FAT ID バイトが、このバッファの最初のバイトです。この場合、ドライバはこのバッファを変更してはなりません。この最初の FAT セクタは、実際の BPB が返される前に読み取られなければならないので、FAT のロケーションは、すべての可能なメディアに関して同じでなければなりません。

NON FAT ID ビットが 1（セット）の場合、ポインタはスクラッチスペース（これは、任意の目的で使用可能）の 1 セクタを指します。BPB の構成方法については、2.8 の“メディアディスクリプタバイト”、および 2.9 の“メディアディスクリプタテーブル”を参照してください。

MS-DOS 3.1 には、ドアロック、または他の手段によってディスクの変更された時期を知らせる機能を持つデバイスに対するサポートが含まれます。これはデバイスドライバによって返される新しいエラーコード (MS-DOS V 3.1 以降でサポートされた：エラー 15) です。このエラーは、“ディスクが変更されてはならない時点で変更された”ことを意味し、ユーザーは、ボリューム ID を用いて正しいディスクを要求されます。ドライバは、リードまたはライト動作で、このエラーを発生させる可能性があります。MS-DOS は、ドライバがメディアの変更を報告し、MS-DOS バッファキャッシュの中に前のディスクへフラッシュする必要のあるバッファがある場合に、MEDIA CHECK でエラーを発生します。

このエラーをサポートするドライバでは、BUILD BPB ファンクションは、ディスクからボリューム ID を読み出すためのトリガとなります。この動作は、ディスクが正常に変更されたことを表します。ボリューム ID は、MS-DOS の FORMAT ユーティリティによってディスクに入れられ、ボリューム ID アトリビュートを持つディスクのルートディレクトリ内の 1 つのエントリとなります。ボリューム ID は、ドライバによって ASCIZ スtring として格納されます。

ドライバがボリューム ID を返さなければならないという要件は、他のボリューム管理のスキームが ASCIZ 文字列を使用している限り、そのスキームを排除するものではありません。NUL（存在しない、またはサポートされていない）ボリューム ID は、慣例により、つぎの String です。

DB "NO NAME" ", 0

2.7.4 リードまたはライト

コマンドコード=3, 4, 8, 9, 12, 16

READ or WRITE (IOCTL も含めて)または, OUTPT UNTIL BUSY

ES : BX →

13 バイト	リクエストヘッダ
1 バイト	BPB からのメディアディスクリプタ
2 ワード	転送アドレス
1 ワード	バイト／セクタカウント
1 ワード	開始セクタ番号 (キャラクタデバイスでは、無視される。)
2 ワード	エラー (0FH) 時、要求されたボリューム ID へのポインタ

ドライバは、セットされたコマンドコードに従って、READ または WRITE コールを実行しなければなりません。ブロックデバイスは、セクタ単位でリードまたはライトし、キャラクタデバイスは、バイト単位でリードまたはライトします。

I/O が完了したら、デバイスドライバはステータスワードをセットし、正常に転送されたセクタ数またはバイト数を報告しなければなりません。エラーによって転送が完了しなかった場合にも、この処理を行う必要があります。エラービットとエラーコードをセットするだけでは、不十分です。

ドライバは、ステータスワードをセットするのに加えて、セクタカウントを実際に転送されたセクタ数(またはバイト数)にセットしなければなりません。IOCTL I/O コールについては、エラーチェックは行われません。デバイスドライバは常に、リターンバイト／セクタカウントを、正常に転送された実際のバイト／セクタ数にセットしなければなりません。

ベリファイスイッチがオンの場合は、デバイスドライバはコマンドコード 9(WRITE WITH VERIFY) で呼び出されます。デバイスドライバは、ライト動作の検証を行うことになります。

ドライバがエラーコード 0FH(無効なデバイス変更)を返す場合は、ドライバは ASCIZ スtring(正しいボリューム ID)に対する DWORD ポインタを返さなければなりません。このエ

ラーコードを返すことによって、MS-DOS に対し、ユーザーがディスクを再挿入するためのプロンプトを出すようトリガがかけられます。デバイスドライバは、BUILD BPB ファンクションの結果、ボリューム ID を読み込んでいなければなりません。

ドライバは、OPEN および CLOSE ファンクションをモニタすることによって、ディスク上のオープンファイルのリファレンスカウントを保守することができます。これによってドライバは、エラー 0FH を返す時期を判定することができます。オープンファイルがまったくなく（リファレンスカウント=0）、ディスクが変更されている場合は、その I/O は成功です。これに対し、オープンファイルがある場合には、0FH が存在している可能性があります。

OUTPUT UNTIL BUSY コールは、プリント待ち行列専用のキャラクタデバイス上の速度を最適化します。デバイスドライバは、デバイスから BUSY を返されるまで、可能な限り、すべてのキャラクタを出力します。環境が整備されていない場合（プリンタが準備されていない等）、デバイスドライバは、このファンクションが実行されないようにしなければなりません。デバイスドライバが、要求されたバイト数より少ないバイト数を出力しても（または、0 バイトを出力しても）、デバイスドライバはそれをエラーとみなさないことに注意してください。

OUTPUT UNTIL BUSY コールを使うことによって、スプーラプログラムは、多くのプリンタが持つ行メモリ（プリンタでは、1 キャラクタずつ受け取って打ち出すのではなく、プリンタの 1 行分か、または改行コードを受け取って 1 ライン分を打ちだします。この 1 ライン分のキャラクタを記憶しておくのが行メモリです）を利用します。

多くのプリンタは、キャラクタの合計が固定されているか、または行数の制限がある受信バッファ（RAM を使用した）を持っています（受信バッファがある場合は、受信バッファから行メモリにキャラクタを渡します）。

プリンタが BUSY を返す前にバッファが一杯になるか、またはキャラクタの途中で BUSY を返すことがあります。バッファは、キャラクタの行を、プリンタにすばやく出力することができますが、それに比べプリンタは印字するのに長い時間を要し、その間プリンタは、BUSY の状態になります。このデバイスコールを使うことによって、バックグラウンドのスプーラプログラムがプリンタ固有のバッファの能力を使うことができます。

プリンタの場合、デバイスドライバ側で、各キャラクタごとにレディ信号まで確認しているとオーバーヘッド（無駄時間）が生じますので、BUSY 信号だけのチェックだけで十分です。

つぎの説明は、ブロックデバイスドライバに適用されます。

ある種の状況において、BIOS が、BIOS I/O パケット内の転送アドレスの“ラップアラウンド”を起こすと思われるような、64K バイトの書き込み動作を実行するよう要求される場合があります。

このリクエストは、MS-DOS のライトコードに追加される最適化処理によって発生するものです。これは、ファイル上の 64K バイトのセクタサイズ内の書き込み動作が、現在の EOF を“通り越してしまう”ことを意味します。このような場合、BIOS は、もしそのように指定する

ならば、“ラップアラウンド”する部分のライト動作を無視することが可能です。

たとえば、転送アドレス×××:1で、1000H バイト相当のセクタの書き込み動作は、最後の2バイトを無視することができます。ユーザープログラムでは、FFFFH バイトを超える I/O をリクエストすることはできず、転送セグメント内で（たとえ 0 でも）ラップアラウンドすることはできません。したがって、この場合は最後の2バイトを無視することができます。

MS-DOS は、2つの FAT を保守しています。DOS が最初の FAT をリードする時に問題があれば、エラーを報告する前に、自動的に2番目の FAT をリードしようとします。BIOS は、すべての再試行に対して責任があります。

COMMAND. COM ハンドラには、自動的な再試行はありませんが、アプリケーションでは、特定のタイプの割り込み 24H エラーに関して、それらを報告する前に、自動的再試行を行う、独自の割り込み 24H ハンドラを持つものがあります。

2.7.5 連続非破壊読み込み

コマンドコード=5

NON DESTRUCTIVE READ NO WAIT ES:BX →

13 バイト リクエストヘッダ
1 バイト デバイスからのリード

つぎに読み込まれた文字を返します。

ステータスワードの DONE ビットがセットされます。

キャラクタデバイスによって BUSY ビット=0（バッファ内に文字が存在します）が返された場合、つぎに読み込まれる文字が返されます。この文字は入力バッファから、削除されません。非破壊読み込み（Non Destructive Read）という用語は、ここからきています。BUSY ビット=1 が返されたときは、バッファ中に文字がありません。

2.7.6 オープンまたはクローズ

コマンドコード=13, 14

OPEN or CLOSE ES:BX →

13 バイト 静的リクエストヘッダ

このファンクションは、V 3.1 以降の MS-DOS で、OPEN/CLOSE/RM のアトリビュートビットがセットされている場合にのみ、サポートされます。デバイス上のカレントファイル

の動作について、デバイスに知らせるように作られています。ブロックデバイス上では、ローカルバッファ管理に使うことができます。デバイスはリファレンスカウンタを保持します。これは、オープンが実行される度にカウンタは1つ加算され、クローズされる度にカウンタは1つ減算されます。カウンタが0になった場合、デバイス上にオープンされているファイルがなく、デバイスは、デバイス内で使用されていた書き込まれたバッファをすべて出力したことになります。

これはデバイスが、交換可能なメディアのディスク上のメディアをユーザーが使用する場合、ユーザーの意志でディスクを交換した場合、エラーを出さないようにするためです。

ブロックデバイス上のこの方法は問題があります。問題点は、FCB コールは、ファイルをクローズすることなしに、オープンできます。それゆえ、メディアを交換しましたかに対して“(Y)”と答えたときに、バッファ内をすべて出力せずに、カウンタが0になった場合、カウンタをリセットし、デバイス側で、BPB の作成のコールを実行するのが賢明です。

これらのコールは、ほとんどキャラクタデバイスで使われます。オープンコールは、デバイスが設定した文字列の先頭から送ることができます。プリンタの場合、フォント(書体)の設定、ページサイズのための特別な制御キャラクタ(文字列)があり、これをデバイス側で処理することができます。

すべてのプロセスが、標準入力、標準出力、標準エラー出力、外部装置、プリンタ(ハンドル 0, 1, 2, 3, 4)にアクセスしてから、CON, AUX, PRN 等の各デバイスは常にオープンされていることを知っていなければなりません。

2.7.7 REMOVABLE MEDIA

コマンドコード=15

REMOVABLE MEDIA ES:BX →

13 バイト 静的リクエストヘッダ

このファンクションは、V 3.1 以降の MS-DOS で、OPEN/CLOSE/RM のアトリビュートビットがセットされている場合にのみ、サポートされます。このコールは、IOCTL システムコールのサブファンクションによって、ブロックデバイスでのみ使用できます。交換不可能なメディア(ハードディスクのような)、または交換可能なメディア(通常のディスクドライブ)のどちらを取り扱うかについてユーティリティに知らせるのに使われます。

ステータスワードの BUSY のビットが返されます。BUSY が 1 ならば、メディアは交換不可能で、BUSY が 0 ならば、交換可能です。エラービットについては、チェックされないことに注意してください。これは、このコールが失敗しないからです。

2.7.8 STATUS

コマンドコード=6, 10

STATUS ES : BX →

13 バイト リクエストヘッダ

このコールは、データが入力または出力を待っているかどうかを示す情報を、MS-DOS に返します。ドライバは、ステータスワードと BUSY ビットを、つぎのようにセットしなければなりません。

キャラクタデバイスへ出力する場合：ドライバがリターン時に、ビット 9 を 1 にセットした場合は、MS-DOS に対し、ライトリクエスト（もし行われていれば）が、現在のリクエストの完了を待っていることを表します。このビットが 0 であれば、現在のリクエストはなく、ライトリクエスト（もし行われていれば）は、すぐに開始されます。

バッファ付きのキャラクタデバイスから入力する場合：ドライバがリターン時にビット 9 を 1 にセットした場合は、バッファリングされているキャラクタはないことを意味し、リードリクエスト（もし行われていれば）が物理的にデバイスに行くことを意味します。0 が返された場合は、デバイスバッファにキャラクタがあり、リードが拒否されないことを意味します。

0 が返された場合は、ユーザーが何かをタイプしたことを意味します。MS-DOS は、すべてのキャラクタデバイスが、入力タイプaheadバッファを持っているものと見なします。タイプaheadバッファを持たないデバイスの場合は、MS-DOS が、存在していないバッファへ何かが入るのを待つことがないように、常に BUSY=0 を返さなければなりません。

2.7.9 FLUSH

コマンドコード=7, 11

FLUSH ES : BX →

13 バイト リクエストヘッダ

FLUSH は、ドライバに対し、ペンディング中のすべてのリクエストをフラッシュ（打ち切り）するよう指示します。このコールは、キャラクタデバイスの入力キューを打ち切る目的で使用します。

デバイスドライバは FLUSH を実行し、ステータスワードをセットし、そしてリターンします。

2.7.10 Generic IOCTL

コマンドコード=19

ES:BX →

13 バイト	静的リクエストヘッダ
バイト	カテゴリ (メジャー) コード
バイト	ファンクション (マイナー) コード
ワード (SI)	の内容
ワード (DI)	の内容
2 ワード	データバッファへのポインタ

このファンクションは、一般的で拡張された IOCTL 機能で、これまでのリード IOCTL とライト IOCTL のデバイスドライバファンクションに代わるために作成されました。MS-DOS 2.0 の IOCTL ファンクションも、IOCTL システムコール (サブファンクション 2, 3, 4, 5) としてサポートされて残っていますが、新しいデバイスドライバは、この Generic IOCTL 機能を使用するとよいでしょう。

Generic IOCTL ファンクションは、カテゴリとファンクションの両方を含んでいます。DOS は、DOS コードによって実際のサービスをするデバイスコマンドであれば横取りするために、カテゴリフィールドを調べます。他のすべてのコマンドカテゴリは、デバイスドライバにサービスさせるために渡されます。

これらのカテゴリとファンクションコードに関するより詳しい解説は、第1章「システムコール」のファンクション 440CH (一般 IOCTL; ハンドル用) と、ファンクション 440DH (一般 IOCTL; ブロックデバイス用) を参照してください。

2.7.11 DEINSTALL

コマンドコード= 20

DEINSTALL ES : BX →

13 バイト	リクエストヘッダ
--------	----------

このファンクションは、インストールされたキャラクタ系デバイスドライバをデインストール(登録されていたデバイスドライバを取り除くこと)するために DELDRV コマンドからコールされます。このファンクションでは、必要に応じて以下のような処理を行う必要があります。

1. キャラクタ系デバイスドライバが、MS-DOS のシステム環境を変更している場合には、変更したシステム環境を元に戻す必要があります。
2. デバイスドライバが ROM BIOS あるいは直接ハードウェアをアクセスし、そのモードなどを変更している場合には、変更したものをもとに戻す必要があります。

注意:

この DEINSTALL ファンクションは、INIT ファンクション(ファンクション 0)にて動的な登録、削除機能をサポートしている旨を宣言している場合にのみコールされます(宣言のしかたについては 2.7.1 "INIT" を参照してください)。

2.7.12 Get/Set Logical Drive Map

コマンドコード=23H (取得) または 24H (設定)

Get/Set Logical Drive Map ES : BX →

13 バイト	静的リクエストヘッダ
バイト	入力 (ユニットコード)
バイト	出力 (最後のデバイス参照)
バイト	コマンドコード
ワード	ステータス
2 ワード	予約

このファンクションは、デバイスヘッダの MS-DOS 3.3 のアトリビュートビットをデバイスドライバがセットしたときだけに、コールされます。このコールは、IOCTL システムコールの

サブファンクションによって、ブロックデバイスのみへ発せられます。マップされた論理ドライブは、ヘッダのユニットコードフィールドでデバイスドライバへ渡されます。デバイスドライバは、要求によってマップされた物理ドライブのオーナーである現在の論理ドライブを返します。

論理ドライブがマップされた物理ドライブを現在所有しているかどうかを検出するために、プログラムはファンクション 440EH または 440FH (論理ドライブマップの取得/設定) をコールした後で、ユニットコードフィールドが変更されていないことを検証 (ベリファイ) する必要があります。

2.8 メディアディスクリプタバイト

MS-DOS では、メディアディスクリプタバイトは、MS-DOS に対し、異なるタイプのメディアが存在することを通知するために使用されます。メディアディスクリプタバイトは、0~FFH の範囲の任意の値をとることができます。

このバイトは、FAT ID バイトと同じである必要はありません。FAT の最初のバイトである FAT ID バイトは、MS-DOS V 2.0 以前では、異なるタイプのディスクメディアを区別する目的で使用されていましたが、V 2.0 以降のディスクデバイスドライバでも、同様に使用することができます。

ただし、FAT ID バイトは、NON FAT ID ビットがセットされない(0)、ブロックデバイスドライバでのみ、意味を持ちます。

メディアディスクリプタバイトまたは FAT ID バイトの値は、MS-DOS に対しては、何の意味も持ちません。これらのバイトは、単にメディアの判定を容易に行えるよう、デバイスドライバに渡されるものです。

重要

BPB コールを行った時に、新しい BPB で返されたメディアバイトが、V 2.0 以前のメディアバイトと同じである場合には、DOS は、そのデバイスに関して、内部構造を再構成しません。MS-DOS は、物理的ディスクが変更された場合にも、フォーマットが変更されていないものとしてディスクを取り扱います。したがって、各々の BPB は、唯一のメディアディスクリプタバイトを持っていなければなりません。

2.9 メディアディスクリプタテーブル

MS-DOS のファイルシステムは、ファイルアロケーションテーブル (FAT) という、ポインタ (各クラスタまたはアロケーションユニットに対する) のリンクされたリストを使用します。

未使用のクラスタは0で表され、エンドオブファイルは FFFH (16 ビットの FAT エントリを持つユニットでは、FFFFH) で表されます。有効なエントリは、0 エントリを指してはなりませんが、もし指した場合には、最初の FAT エントリ (0 エントリによって指される) は予約され、エンドオブチェーンにセットされます。その結果、いくつかのエンドオブチェーン値が定義されており ((F)FF8~(F)FFF)、これらは異なるメディアのタイプを区別するために使用されています。

最善の方法は、ブートセクタに完全なメディアディスクリプタテーブルを書き、それをメディアの識別に使用することです。NON FAT ID ビットをセットしないドライバを持つシステムの将来の MS-DOS のバージョンに対する互換性を保証するために、FORMAT のプロセス中に FAT ID バイトを書き込むことも必要です。

今後、さまざまなディスクフォーマットをサポートするための柔軟性を高めるためには、特定の種類のメディアについての BPB に関する情報を、ブートセクタに保存しておくことが望ましいと言えます。このようなブートセクタのフォーマットを、つぎに示します。

↓ B P B	3 バイト	ブートコードへの near jump
	8 バイト	メーカー名およびバージョン
	1 ワード	1 セクタあたりのバイト数
	1 バイト	1 アロケーションユニットあたりのセクタ数
	1 ワード	予備のセクタ数
B P B ↑	1 バイト	FAT の数
	1 ワード	ルートディレクトリエントリの数
	1 ワード	論理イメージ内のセクタ数
	1 バイト	メディアディスクリプタ
	1 ワード	1 FAT セクタ数
	1 ワード	1 トラックあたりのセクタ数
	1 ワード	ヘッド数
	1 ワード	隠れたセクタの数

最後の3つのワード(“トラック当りのセクタ数”, “ヘッド数”, および“隠れたセクタの数”)は, MS-DOS によっては使用されませんが, デバイスドライバで使用することができます。これらは, デバイスドライバがメディアを理解するための補助手段です。“トラック当りのセクタ数”および“ヘッド数”は, 論理的レイアウトが同じで物理的レイアウトが異なるメディア(例: 40トラック両面と, 80トラック片面)をサポートする場合に有益です。“トラック当りのセクタ数”は, デバイスドライバに対し, 物理的なディスク上に, 論理的ディスクフォーマットがどのようにレイアウトされているかを指示します。“隠れたセクタの数”は, ドライブのパーティション(区画化)スキーマをサポートするため使用することができます。

NON FAT ID フォーマットドライバによるメディア判定には, つぎのプロシージャが適当です。

1. ドライブのブートセクタを, DWORD 転送アドレスによって指定される1セクタのスクラッチスペースに読み込みます。
2. ブートセクタの最初のバイトが, E9H または EBH (3 バイト NEAR または 2 バイト SHORT JUMP の最初のバイト) か, あるいは EBH (後に NOP が続く 2 バイト JUMP の最初のバイト) のいずれかであることを判定します。もしそうならば, BPB はオフセットを基点として置かれます。それに対するポインタを返します。
3. ブートセクタに BPB テーブルがない場合には, それは MS-DOS の V 2.0 以前でフォーマットされたディスクであり, おそらく FAT ID バイトを用いてメディアを判定します。

ドライバはオプションとして, FAT の最初のセクタを, 1セクタのスクラッチ領域に読み込み, 最初のバイトを読むことによってメディアタイプを判定することができます。この判定はシステムによって読まれるべきディスク上で使用されている FAT ID バイトに基づくものとします。ハードコードッド BPB に対するポインタを返します。

2.10 クロックデバイス

よく用いられる追加ボードに、リアルタイムクロックボードがあります。時刻および日付を得るために、このボードをシステム内に統合できるようにする（アトリビュートワードによって決定される）クロックデバイスという特殊な装置があります。クロックデバイスは、他のすべてのキャラクタデバイスと同様にファンクションを定義し、それを実行します。ファンクションの大部分は、“DONE ビットをセットし、エラービットをリセットし、リターンする”というものです。このデバイスに対してリード/ライトが行われると、6 バイトの値が転送されます。先頭の 2 バイトは、1980 年 1 月 1 日から数えた日数のワード、3 バイト目は分、4 バイト目は時、5 バイト目は 1/100 秒、および 6 バイト目は秒を示します。クロックデバイスリードによって日付および時刻を取得し、ライトによって日付および時刻をセットします。

2.11 デバイスコールの分析

MS-DOS が、ライトリクエストを実行するためにブロックデバイスドライバを呼び出した時の動作を具体的に説明します。

1. MS-DOS は、メモリの予約された領域に、リクエストパケットを書き込みます。
2. MS-DOS は、ブロックデバイスドライバの、ストラテジエントリポイントと呼び出します。
3. デバイスドライバは、ES および BX レジスタをセーブし (ES:BX は、リクエストパケットを指します)、そして FAR リターンを行います。
4. MS-DOS は、割り込みエントリポイントと呼び出します。
5. デバイスドライバは、リクエストパケットへのポインタを取得し、コマンドコード（オフセット 2）を読んで、これがライトリクエストであることを判定します。デバイスドライバは、このコマンドコードをディスパッチテーブルへの索引に変換し、制御はディスクライトルーチンに渡されます。
6. デバイスドライバは、ユニットコード（オフセット 1）を読んで、どのディスクドライブでライト動作を行うかを判定します。
7. コマンドはディスクライトなので、デバイスドライバは、リクエストパケットから、転送アドレス（オフセット 14）、セクタカウント（オフセット 18）、およびスタートセクタ（オフセット 20）を入手しなければなりません。
8. デバイスドライバは、最初の論理セクタ番号をトラック、ヘッド、およびセクタ番号に変換します。

9. デバイスドライバは、ユニットコード（このデバイスドライバによって定義されるサブユニット）で定義されるドライブの指定された開始セクタから初めて、指定された数のセクタを書き込み、リクエストパケットで示される転送アドレスからデータを転送します。この中には、ディスクコントローラに対する複数のライトコマンドが含まれていてもさしつかえありません。
10. 転送が完了したら、デバイスドライバは、ステータスワード（リクエストパケットのオフセット 3）の中の DONE ビットをセットすることによって、MS-DOS に対し、リクエストのステータスを報告しなければなりません。リクエストパケットのセクタカウント領域に、実際に転送されたセクタ数を報告します。
11. エラーが発生した場合は、ドライバはステータスワード内の DONE ビットおよびエラービットをセットし、ステータスワードの下半分にエラーコードを書き込みます。実際に転送されたセクタ数を、リクエストヘッダに書き込まなければなりません。ステータスワードのエラービットをセットするだけでは、不十分です。
12. デバイスドライバは、MS-DOS へ FAR リターンを行います。

デバイスドライバは、MS-DOS の状態を保存しなければなりません。これは、すべてのレジスタ（フラグも含めて）を保存しなければならないことを意味します。方向フラグと割り込みイネーブルビットは重要です。デバイスドライバの割り込みエントリポイントが呼び出される時点で、MS-DOS は、内部スタックに約 40～50 バイトの余地を持っています。デバイスドライバが、拡張的なスタック動作を使用する場合は、ローカルスタックに切り換えなければなりません。

2.12 デバイスドライバ例

以下に、ブロックデバイスドライバおよびキャラクタデバイスドライバのプログラム例を掲載します。

なお、プログラム例は、そのままではアセンブルして動作可能なものではありません。

2.12.1 ブロックデバイスドライバ

```

;*****
;*
;*      デバイスドライバ      (ブロックデバイス)
;*
;*****

```

;このプログラムは、ブロックデバイスの例です

```

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE

```

```

;*****
;
;      デバイスヘッダ
;

```

;デバイスヘッダは、次の形式です

```

;-----+-----+-----+
; | 次のデバイスヘッダへのポインタ          | ダブルワード |
; |-----+-----+-----+
; | 最後のデバイスヘッダの場合は、-1 とする |
; |-----+-----+-----+
; | アトリビュート                          | ワード          |
; |-----+-----+-----+
; | 1 5 ビット目                            |
; |      1      = キャラクタデバイス
; |      0      = ブロックデバイス
; | 1 5 ビット目が 1 (キャラクタデバイス) の場合
; | 0 ビット目 1 ならば、標準入力用デバイス
; | 1 ビット目 1 ならば、標準出力用デバイス
; | 2 ビット目 1 ならば、ヌルデバイス (DOS で使用)
; | 3 ビット目 1 ならば、クロックデバイス
; | 4 ビット目 - 1 2 ビット目
; | 全て 0 にする
; | 1 3 ビット目 (ブロックデバイスのみ)
; | 0 ならば、IBM フォーマット
; | 1 4 ビット目
; | 1 ならば、コントロールストリングの処理可能
; |-----+-----+-----+
; | ストラテジルーチンへのエントリポイント | ワード          |
; |-----+-----+-----+
; | 割込みルーチンへのエントリポイント    | ワード          |
; |-----+-----+-----+
; | 名前                                    | 8 バイト        |
; |-----+-----+-----+
; |      キャラクタデバイスの場合は、8 文字でデバイスの名前
; |      を指定する
; |      ブロックデバイスの場合は、先頭の 1 バイトでユニット
; |      数を指定する
; |-----+-----+-----+

```



```

DSKDEV LABEL WORD
      DD -1 ;最後のデバイス
      DW 0000H ;ブロック・I B Mフォーマット
      DW STRATEGY ;ストラテジエントリポイント
      DW DSK_INT ;割込みエントリポイント
DSKNUM DB 2 ;ユニット数

```

;このテーブルは、コマンドを割振るためのものです

```

DSKTBL LABEL WORD
      DW DSK_INIT ;INIT
      DW MEDIA_CHK ;MEDIA CHECK
      DW GET_BPB ;BUILD BPB
      DW CMD_ERR ;IOCTL INPUT
      DW DSK_READ ;INPUT
      DW EXIT ;NON-DESTRUCTIVE INPUT NO WAIT
      DW EXIT ;INPUT STATUS
      DW EXIT ;INPUT FLUSH
      DW DSK_WRT ;OUTPUT
      DW DSK_WRTV ;OUTPUT & VERIFY
      DW EXIT ;OUTPUT STATUS
      DW EXIT ;OUTPUT FLUSH
      DW EXIT ;IOCTL OUTPUT

```

PAGE

```

;*****
;
; ストラテジルーチン
;

```

```

PTRSAV DD 0 ;リクエストヘッダアドレス退避用

```

```

STRATP PROC FAR
STRATEGY:
      MOV WORD PTR CS:[PTRSAV],BX ;リクエストヘッダアドレスの退避
      MOV WORD PTR CS:[PTRSAV+2],ES
      RET
STRATP ENDP

```

PAGE

```

;*****
;
; 割込みルーチン
;
;
; リクエストヘッダの定義

```

```

REQ_HEAD STRUC
CMDLEN DB ? ;コマンド長
UNIT DB ? ;ユニットコード
CMD DB ? ;コマンドコード
STATUS DW ? ;ステータス
      DB 8 DUP(?)
MEDIA DB ? ;メディアディスクリブタバイト
TRANS DD ? ;転送アドレス
COUNT DW ? ;セクタカウント
START DW ? ;開始セクタ番号
REQ_HEAD ENDS

```

```

; 割込みルーチン

```

```

DSK_INT:
      PUSH SI ;レジスタの退避
      PUSH AX
      PUSH CX
      PUSH DX
      PUSH DI
      PUSH BP
      PUSH DS
      PUSH ES
      PUSH BX

```

```

LDS      BX,CS:[PTRSAV]      ;リクエストヘッダアドレスの取得

MOV      AL,[BX.UNIT]        ;ユニットコードの取得
MOV      AH,[BX.MEDIA]      ;メディアディスクリブタバイトの取得
MOV      CX,[BX.COUNT]      ;セクタカウントの取得
MOV      DX,[BX.START]      ;開始セクタ番号の取得
XCHG     AX,DI               ;レジスタAXの退避
MOV      AL,[BX.CMD]        ;コマンドコードの取得
CMP      AL,11
JA       CMD_ERR             ;コマンドエラー
XOR      AH,AH
SHL      AX,1
MOV      SI,OFFSET DSKTBL
ADD      SI,AX
XCHG     AX,DI               ;レジスタAXの復帰
LES      DI,[BX.TRANS]       ;転送アドレスの取得
PUSH     CS
POP      DS                  ;DS ← CS
JMP      [SI]                ;コマンド処理ルーチンへ

```

PAGE

```

;*****
;
;      出口
;

```

```

CMD_ERR:      MOV      AL,3      ;コマンドエラー
;              ;エラーコードのセット

ERR_EXIT:      MOV      AH,10000001B ;エラーリターン
;              ;エラービット、ダンビットのセット
JMP          SHORT EXIT1

EXITP PROC FAR
EXIT:      MOV      AH,00000001B ;ノーエラーリターン
;              ;ダンビットのセット

EXIT1:      LDS      BX,CS:[PTRSAV] ;リクエストヘッダアドレスの取得
MOV         [BX.STATUS],AX ;ステータスのセット

POP      BX ;レジスタの復帰
POP      ES
POP      DS
POP      BP
POP      DI
POP      DX
POP      CX
POP      AX
POP      SI
RET

EXITP ENDP

```

PAGE

```

;*****
;
;      コマンド処理ルーチン
;

```

```

;*****
;
;      MEDIA_CHK:      ;MEDIA CHECK
LDS      BX,[PTRSAV]    ;リクエストヘッダアドレスの取得
MOV      BYTE PTR [BX.TRANS],0 ;変更されたかわからない
JMP      EXIT

```

```

;*****
;
GET_BPBP:                                ;BUILD BPB
      MOV     AH,ES:[DI]                  ;F A T   I D   バイトの取得
      MOV     SI,OFFSET DSKBPB           ;9 セクタ/トラックをセット
      CMP     AH,0F9H                    ;F A T   I D   は9 セクタ/トラックか
      JE      DSK_9                      ;8 セクタ/トラックをセット
      MOV     SI,OFFSET DSKBPB1
DSK_9:  LDS     BX,[PTRSAV]                ;リクエストヘッダアドレスを取得
      MOV     [BX.MEDIA],AH              ;メディアディスクリプタをセット
      MOV     [BX.COUNT],SI              ;B P B へのポインタをセット
      MOV     [BX.COUNT+2],CS
      JMP     EXIT

;*****
;
DSK_READ:                                ;INPUT
      MOV     RW_FLG,0                   ;リードをセット
      MOV     VRFY_FLG,0
RW_COMMON:
      JCXZ    NO_IO                      ;セクタカウントが0なら、何もしない
      CALL    RW_EXE
      PUSHF
      LDS     BX,[PTRSAV]                ;リクエストヘッダアドレスを取得
      SUB     [BX.COUNT],CX              ;転送したセクタ数のセット
      POPF
      JC      NO_IO                      ;キャリーフラグの復帰
      JC      ERR_EXIT                   ;C F   =   0   ノーエラー
      JC      ERR_EXIT                   ;      =   1   エラー
NO_IO:  JMP     EXIT

;
DSK_WRT:                                ;OUTPUT
      MOV     RW_FLG,1                   ;ライトをセット
      MOV     VRFY_FLG,0                 ;ベリファイをオフ
      JMP     RW_COMMON

;
DSK_WRTV:                                ;OUTPUT & VERIFY
      MOV     RW_FLG,1                   ;ライトをセット
      MOV     VRFY_FLG,1                 ;ベリファイをオン
      JMP     RW_COMMON

PAGE

;*****
;
      入力:
      AL = ユニットコード
      AH = メディアディスクリプタ
      CX = セクタカウント
      DX = 開始セクタ番号
      ES:DI = 転送アドレス
      出力:
      成功: キャリーフラグ = 0
      失敗:                  = 1
              AL = エラーコード
              CX = 転送されなかったセクタカウント
;
RW_EXE:
      MOV     UNIT_NO,AL                  ;ユニットコードの退避
      MOV     SEC_CNT,CX                  ;セクタカウントの退避
      CALL    SET_UP                      ;開始セクタ番号 --> 物理セクタ番号
LOOP_RW:
      CALL    RW
      JC      RW_ERR
      CALL    SET_UP1                     ;次のセクタにセット
      DEC     SEC_CNT
      JNZ     LOOP_RW
RW_ERR:
      MOV     CX,SEC_CNT                  ;残りのセクタ数
      RET

```



```

;*****
;
; 入力:
; 出力:
;       D X = 開始セクタ番号
;       B X = 転送する長さ (バイト)
;       C H = セクタ長
;       C L = シリンダ番号
;       D H = ヘッド番号
;       D L = セクタ番号
;
SET_UP:
MOV     BL,9                ;9セクタ/トラック
CMP     AH,0F9H            ;メディアディスクリブタバイトのチェック
JE      DSK9_01
MOV     BL,8                ;8セクタ/トラック
DSK9_01:
MOV     SEC_TRK,BL
MOV     AX,DX               ;A X ← 開始セクタ番号
DIV     BL
INC     AH
MOV     DL,AH              ;セクタ番号のセット
XOR     DH,DH
SHR     AL,1
RCL     DH,1               ;ヘッド番号のセット
MOV     CL,AL              ;シリンダ番号のセット
MOV     BX,512             ;転送長のセット
MOV     CH,02H            ;セクタ長のセット
;0: 128バイト   1: 256バイト
;2: 512バイト   3: 1024バイト
RET

;
; SET_UP1:
;       INC     DL                ;次のセクタ
;       CMP     DL,SEC_TRK        ;次のトラックか
;       JBE     NOT_NXT
;       MOV     DL,1              ;セクタ #1 をセット
;       XOR     DH,1              ;ヘッドを反転
;       JNZ     SAME_CYL
;       INC     CL                ;次のシリンダへ
;
SAME_CYL:
NOT_NXT:
ADD     DI,BX               ;バッファのアップデート
RET

;
; RW:
;       CMP     RW_FLG,0          ;リードか
;       JNE     WRT
;       ; ここで、リードをする
;       ; エラーが起これたら、キャリーフラグをセット
;       ;
;       JC      ERROR_SET        ;エラーならジャンプ
;       RET
;
WRT:
;       ; ここで、ライトをする
;       ; エラーが起これたら、キャリーフラグをセット
;       ;
;       JC      ERROR_SET        ;エラーならジャンプ
;       CMP     VRFY_FLG,0       ;ベリファイオフか
;       JE      NOT_VRFY
;       ;
;       ; ここで、ベリファイをする
;       ; エラーが起これたら、キャリーフラグをセット

```

```

:
: JC          ERROR_SET          ;エラーならジャンプ
NOT_VRFY:
: RET

ERROR_SET:
: ; ここで、レジスタ AL にエラーコードをいれる
:
: STC
: RET

:
: データ
:

RW_FLG      DB      ?           ; 0 : リード
:                               ; 1 : ライト
VRFY_FLG     DB      ?           ; 0 : ベリファイオフ
:                               ; 1 : ベリファイオン
UNIT_NO      DB      ?           ; ユニットコード
SEC_TRK      DB      ?           ; セクタ数ノトラック
SEC_CNT      DW      ?           ; セクタカウント

PAGE

:*****
:
: B P B の定義
:
DSKBPB:
:                               ; 5 インチ 2 D D (9 セクタノトラック)
:                               ; 物理セクタ長 (バイト)
:                               ; セクタ数ノアロケーションユニット
:                               ; リザーブセクタ数
:                               ; F A T のコピー数
:                               ; ディレクトリエントリ数
:                               ; セクタ数ノボリューム
:                               ; メディアディスクリブタバイト
:                               ; セクタ数ノ F A T
DW      512
DB      1
DW      1
DB      2
DW      112
DW      9*2*80
DB      0F9H
DW      3

DSKBPB1:
:                               ; 5 インチ 2 D D (8 セクタノトラック)
:                               ;
:                               ;
:                               ;
:                               ;
:                               ;
:                               ;
:                               ;
DW      512
DB      1
DW      1
DB      2
DW      112
DW      8*2*80
DB      0FBH
DW      2

INIT_TBL:
:                               ; ユニット # 0
:                               ; ユニット # 1
DW      OFFSET DSKBPB
DW      OFFSET DSKBPB1

:*****
:
DSK_INIT:
:                               ; INIT
:                               ; ユニット数の取得
:                               ; レジスタ DS の退避
:                               ; リクエストヘッダアドレスの取得
:                               ; ユニット数のセット
MOV      AL, DSKNUM
PUSH     DS
LDS      BX, [PTRSAV]
MOV      [BX.MEDIA], AL
MOV      WORD PTR [BX.TRANS], OFFSET DSK_INIT
MOV      WORD PTR [BX.TRANS+2], CS
MOV      WORD PTR [BX.COUNT+2], CS ; ブレークアドレスのセット
MOV      WORD PTR [BX.COUNT], OFFSET INIT_TBL
:                               ; B P B 配列アドレスのセット
:                               ; レジスタ DS の復帰
POP      DS
JMP      EXIT

CODE     ENDS
END

```

2.12.2 キャラクタデバイスドライバ

以下はキャラクタデバイスドライバのプログラム例です。

***** キャラクタデバイスドライバのプログラム例 *****

```

CODE    SEGMENT BYTE

        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE

;      EQU

CR       EQU     13           ; キャリッジリターン・コード
BACKSP   EQU     8           ; バックスペース・コード
_ESC     EQU     1BH         ; エスケープ・コード
BASE     EQU     0A000H      ; VRAMセグメント

;-----
;
;      CON      コンソール・デバイスドライバの例
;
CONDEV    DW      -1.-1
          DW      10000000000000011B      ; CON_IN と CON_OUT の サポート
          DW      STRATEGY
          DW      ENTRY
          DB      'CON'

;-----
;
;      コマンド ジャンプ テーブル
;
CONTBL    DW      CON_INIT      ; 初期設定
          DW      EXIT          ; 媒体検査
          DW      EXIT          ; BPB作成(フロッピーのみ)
          DW      CMDERR        ; IOCTL入力
          DW      CON_READ      ; 入力(入力待ち)
          DW      CON_RDND      ; 連続非破壊読み込み
          DW      EXIT          ; 入力状況
          DW      CON_FLSH      ; 入力要求終了
          DW      CON_WRIT      ; 出力(書き込み)
          DW      CON_WRIT      ; 出力(書き込み)
          DW      EXIT          ; 出力状況
          DW      EXIT          ; 出力要求終了
          DW      EXIT          ; IOCTL出力

CMDTABL    DB      'A'
          DW      CUU           ; CURSOR UP
          DB      'B'
          DW      CUD           ; CURSOR DOWN
          DB      'C'
          DW      CUF           ; CURSOR FORWARD
          DB      'D'
          DW      CUB           ; CURSOR BACK
          DB      'H'
          DW      CUH           ; CURSOR POSITION
          DB      'Y'
          DW      CUP           ; CURSOR POSITION
          DB      'j'
          DW      PSCP          ; SAVE CURSOR POSITION
          DB      'k'
          DW      PRCP          ; RESTORE CURSOR POSITION
          DB      00

```



```

PAGE
-----
:
:   割込みルーチン
:
:
CMDLEN EQU 0           ;コマンド長
UNIT EQU 1             ;ユニットコード
CMD EQU 2              ;コマンドコード
STATUS EQU 3           ;ステータス
MEDIA EQU 13           ;
TRANS EQU 14           ;転送アドレス
COUNT EQU 18
START EQU 20

PTRSAV DD 0

STRATP PROC FAR
:
:   ストラテジー
:
STRATEGY:
    MOV     WORD PTR CS:[PTRSAV],BX    ;リクエストヘッダのオフセットを保管
    MOV     WORD PTR CS:[PTRSAV+2],ES  ;セグメントを保管
    RET

STRATP ENDP
:
:   共通入口点
:
ENTRY:
    PUSH    SI
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    DI
    PUSH    BP
    PUSH    DS
    PUSH    ES
    PUSH    BX

    LDS     BX,CS:[PTRSAV]             ;リクエストヘッダアドレスの取得
    MOV     CX,WORD PTR [BX,COUNT]     ;バイトカウンタの取得
    MOV     AL,BYTE PTR [BX,CMD]       ;コマンドコードの取得
    CBW
    MOV     SI,OFFSET CONTBL
    ADD     SI,AX
    ADD     SI,AX
    CMP     AL,12
    JA      CMDERR
    LES     DI,[BX,TRANS]
    PUSH    CS
    POP     DS

    JMP     WORD PTR[SI]

PAGE
-----
:
:   出口
:
:
BUS_EXIT:
    MOV     AH,00000011B              ;ビジービット, 処理済みビットをセット
    JMP     SHORT ERR1

CMDERR:
    MOV     AL,3

```

```

ERR_EXIT:
    MOV     AL,10000001B      ;エラービット，処理済みビットをセット
    JMP     SHORT ERR1

EXITP    PROC    FAR

EXIT:
    MOV     AH,00000001B      ;処理済みビットをセット
ERR1:
    LDS     BX,CS:[PTRSAV]
    MOV     WORD PTR [BX.STATUS],AX

    POP     BX
    POP     ES
    POP     DS
    POP     BP
    POP     DI
    POP     DX
    POP     CX
    POP     AX
    POP     SI
    RET

EXITP    ENDP

-----
:
:      コンソール出力サブルーチン
:
STATE    DW      SI
MAXCOL   DB      79
COL       DB      0          ;カラム
ROW       DB      0          ;ライン
SAVCR    DW      0          ;カーソル位置セーブ用

ATTR     DB      11100001B   ;文字アトリビュート
:
:      VRAMアドレス・テーブル
:
LINE_TABLE LABEL WORD
    DW      0000H            ;0-LINE
    DW      00A0H            ;1
    DW      0140H            ;2
    DW      01E0H            ;3
    DW      0280H            ;4
    DW      0320H            ;5
    DW      03C0H            ;6
    DW      0460H            ;7
    DW      0500H            ;8
    DW      05A0H            ;9
    DW      0640H            ;10
    DW      06E0H            ;11
    DW      0780H            ;12
    DW      0820H            ;13
    DW      08C0H            ;14
    DW      0960H            ;15
    DW      0A00H            ;16
    DW      0AA0H            ;17
    DW      0B40H            ;18
    DW      0BE0H            ;19
    DW      0C80H            ;20
    DW      0D20H            ;21
    DW      0DC0H            ;22
    DW      0E60H            ;23
    DW      0F00H            ;24

```

```

CHROUT:
    CMP     AL,13
    JNZ     TRYLF
    MOV     [COL],0
                                ;キャリッジリターン処理
                                ;カラムを0にセット
    JMP     SHORT SETIT

TRYLF:
    CMP     AL,10
                                ;LFか ?
    JZ      LF
    CMP     AL,7
    JNE     TRYBACK
                                ;ブザー処理
    MOV     AL,06H
    OUT     37H,AL
                                ;ブザーON
    MOV     CX,6000H

BEL1:
    PUSH    AX
    POP     AX
    LOOP    BEL1

    MOV     AL,07H
    OUT     37H,AL
                                ;ブザーOFF
RET5:    RET

TRYBACK:
    CMP     AL,8
                                ;BSか ?
    JNZ     OUTCHR
    CMP     [COL],0
                                ;バックスペース処理
    JZ      RET5
    DEC     [COL]
    JMP     SHORT SETIT

;
; 文字の表示
;
OUTCHR:
    XOR     AH,AH
    MOV     DH,[ROW]
    MOV     DL,[COL]
    CALL    LCCONV
                                ;ライン,カラムからVARMアドレスを得る
    MOV     DI,BX
    MOV     DX,[BASE]
    MOV     ES,DX
    MOV     ES:[DI],AX
                                ;文字のセット
    MOV     AL,[ATTR]
    ADD     DI,2000H
                                ;アトリビュートのセット
    STOSW
    INC     [COL]
    MOV     AL,[COL]
    CMP     AL,[MAXCOL]
    JBE     SETIT
    MOV     [COL],0

LF:
                                ;ラインフィード処理
    INC     [ROW]
    CMP     [ROW],24
                                ;現在行に1加えて
                                ;24以上になったらスクロールアップする
    JB      SETIT
    MOV     [ROW],23
    CALL    SCROLL

SETIT:
                                ;カーソル表示処理
    MOV     DH,ROW
    MOV     DL,COL
    CALL    LCCONV
    SHR     BX,1
    CLI

```


SETIT1:

```

IN      AL,60H
TEST    AL,04H
JZ      SETIT1
MOV     AL,49H
OUT     62H,AL
MOV     AL,BL
OUT     60H,AL
MOV     AL,BH
OUT     60H,AL
STI
RET

```

SCROLL:

:スクロールアップ処理

```

CLD
PUSH    DS
MOV     AX,[BASE]
MOV     ES,AX
MOV     DS,AX
XOR     DI,DI
MOV     SI,80*2
MOV     CX,23*80
REP     MOVSW
MOV     DI,2000H
MOV     SI,2000H+80*2
MOV     CX,23*80
REP     MOVSW
MOV     DH,23
XOR     DL,DL
CALL    LCCONV
MOV     DI,BX
MOV     AL,' '
MOV     CX,80
REP     STOSW
MOV     AL,[ATTR]
MOV     CX,80
MOV     DI,BX
ADD     DI,2000H
REP     STOSW
POP     DS
RET

```

:コードデータ移送

:アトリビュートデータ移送

:最終行スペースクリア

LCCONV:

:ライン, カラムからVRAMアドレスへの変換

```

XOR     BX,BX
MOV     BL,DH
SHL     BX,1
ADD     BX,OFFSET LINE_TABLE
MOV     BX,CS:[BX]
XOR     DH,DH
SHL     DL,1
ADD     BX,DX
RET

```

```

:-----
:
:
:

```

:コンソール入力ルーチン

CON_READ:

JCXZ CON_EXIT

CON_LOOP:

```

PUSH    CX
CALL    CHRIN
POP     CX
STOSB
LOOP    CON_LOOP

```

:ALに人力文字を得る

```
CON_EXIT:
    JMP     EXIT
```

```
-----
:
:      コンソール入力サブルーチン
:
```

```
CHRIN:
    MOV     AH,1                ;キーボード入力チェック
    INT     18H
    CMP     BH,1
    JNE     CHRIN
    MOV     AH,0                ;入力文字の引き取り
    INT     18H
    RET
```

```
-----
:
:      連続非破壊読み込み
:
```

```
CON_RDND:
    MOV     AH,1                ;キーボード入力チェック
    INT     18H
    CMP     BH,1
    JNE     CONBUS              ;入力なし
RDEXIT:
    LDS     BX,[PTRSAV]
    MOV     [BX.MEDIA],AL
EXVEC:    JMP     EXIT
CONBUS:    JMP     BUS_EXIT      ;入力なし(ビジー)
```

```
-----
:
:      取り消しコール
:
```

```
CON_FLSH:
    MOV     AH,3
    INT     18H                ;キーボード初期化
    JMP     EXVEC
```

```
-----
:
:      出力(書き込み)
:
```

```
CON_WRIT:
    JCXZ     EXVEC
CON_LP:
    MOV     AL,ES:[DI]          ;出力文字 -> AL
    INC     DI
    CALL    OUTC
    LOOP    CON_LP
    JMP     EXVEC
OUTC:
    PUSH    AX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSH    DI
    PUSH    ES
    PUSH    BP
    CALL    VIDEO
```

```

POP    BP
POP    ES
POP    DI
POP    SI
POP    DX
POP    CX
POP    AX
RET

```

```

:-----
:
:
:
VIDEO:
MOV    SI,OFFSET STATE
JMP    [SI]

S1:
CMP    AL,_ESC                ;エスケープシーケンス ?
JNE    S1B
MOV    WORD PTR [SI],OFFSET S2 ;ESCシーケンス処理ルーチンのアドレス
RET

S1B:   CALL    CHROUT
S1A:   MOV     WORD PTR [STATE],OFFSET S1
RET

:
:   ESCシーケンス解析ルーチン
:
S2:
MOV    BX,OFFSET CMDTABL-3

S7A:
ADD    BX,3
CMP    BYTE PTR[BX],0
JZ     S1A
CMP    [BX],AL
JNE    S7A
JMP    WORD PTR[BX+1]        ;該当サブルーチンへ飛ぶ

MOVCUR:
CMP    [BX],AH
JE     SETCUR
ADD    [BX],AL

SETCUR:
CALL   SETIT
JMP    S1A

CUP:
MOV    WORD PTR[SI],OFFSET CUP1 ;カーソルアドレッシング
RET

CUP1:
SUB    AL,32
MOV    [ROW],AL
MOV    WORD PTR[SI],OFFSET CUP2
RET

CUP2:
SUB    AL,32
MOV    [COL],AL
JMP    SETCUR

SM:
MOV    WORD PTR[SI],OFFSET S1A
RET

```


CUH:	MOV JMP	WORD PTR COL,0 SETCUR	:カーソルをホーム位置へ移動する
CUF:	MOV MOV	AH,MAXCOL AL,1	:カーソルを右へ移動する
CUF1:	MOV JMP	BX,OFFSET COL MOVCUR	
CUB:	MOV JMP	AX,00FFH CUF1	:カーソルを左へ移動する
CUU:	MOV	AX,00FFH	:カーソルを上へ移動する
CUU1:	MOV JMP	BX,OFFSET ROW MOVCUR	
CUD:	MOV JMP	AX,23*256+1 CUU1	:カーソルを下へ移動する
PSCP:	MOV MOV JMP	AX,WORD PTR COL SAVCR,AX SETCUR	:現在のライン, カラムを退避
PRCP:	MOV MOV JMP	AX,SAVCR WORD PTR COL,AX SETCUR	:退避したライン, カラムを復帰

```

:
: 初期設定
:
CON_INIT:
    PUSH    DS
    LDS     BX,[PTRSAV]
    MOV     WORD PTR[BX.TRANS],OFFSET CON_INIT
    MOV     WORD PTR[BX.TRANS+2],CS
    POP     DS
    JMP     EXIT

CODE     ENDS
END

```

第3章

MS-DOS技術資料

3.1 MS-DOS の初期化

MS-DOS の初期化は、つぎのようなステップで行われます。まず、ROM（リードオンリーメモリ）ブートストラップに制御が渡され、つぎにこのブートストラップによってディスクからブートセクタが読み込まれます。つづいてこのブートセクタによって、つぎのファイルが読み込まれます。

IO.SYS

MSDOS.SYS

これらのファイルが読み込まれると、ブート処理を開始します。

3.2 コマンドプロセッサ

MS-DOS コマンドプロセッサ (COMMAND.COM) は、常駐部、初期化部、非常駐部の3つの部分から構成されています。

1. 常駐部は、MSDOS.SYS とそのデータ域のすぐ後のメモリ中に存在します。この部分は、割り込みタイプ 22H（終了アドレス）、23H（CTRL-C 抜け出しアドレス）、24H（致命的なエラーによる打ち切りアドレス）を処理するためのルーチン、および必要な場合非常駐部をロードするためのルーチンから構成されています（プログラムの終了時、チェックサム方式によってプログラムが非常駐部にオーバーレイが行われたか調べます。オーバーレイが行われた場合は再ロードを行います）。すべての標準 MS-DOS エラーハンドリングは、COMMAND.COM のこの部分で行われます。このハンドラには、エラーメッセージのスクリーン出力および“中止〈A〉、もう一度〈R〉、無視〈I〉?”の応答の解釈ルーチンが含まれています。
2. 初期化部は常駐部のつぎに存在し、開始時に制御が渡されます。この部分には AUTOEXEC.BAT ファイルの処理ルーチンが入っています。プログラムのロード可能なセグメントアドレスは、初期化部分によって決定されます。それ以後は必要ないので、最初にロードされる COMMAND.COM のプログラムによってオーバーレイされます。

3. 非常駐部は、メモリの最上位にロードされます。この部分には、すべての内部コマンドとバッチファイルプロセッサが入っています。

コマンドプロセッサの3番目の部分によって、プロンプト(A>のような)が表示されコマンドのキーボード(またはバッチファイル)入力と実行が行われます。外部コマンドの場合、コマンドラインを作成し、プログラムのロードと制御の移行を行うためのEXECファンクションコール(ファンクション4BH, コード00H)が行われます。

3.3 MS-DOS ディスクアロケーション

MS-DOSのディスクスペースは、以下のようなフォーマットになっています。領域のサイズはいずれも可変です。

予備領域

ファイルアロケーションテーブル(1)

ファイルアロケーションテーブル(2)

ルートディレクトリ

データ領域

種々ファイルのためのスペース(データ領域)は、必要な場合に割り当てられ、前もって割り当てられているものではありません。スペースは、一度に1クラスタ(アロケーションの単位)ずつ割り当てられます。クラスタとは、常に連続したいくつかのセクタのことで、クラスタは、ファイルアロケーションテーブル(FAT)を通して“連結”されています。また、1クラスタの中のセクタ数は必ず2の累乗です。また信頼性を高めるために、最初のFATのコピーである2番目のFATが保存されています。また第1のFATの途中でスキップセクタが発生した場合でも2番目のFATを使用することができ、使用不可のディスクでもデータの回復をすることができます。

3.4 MS-DOS ディスクディレクトリ

FORMAT コマンドは、すべてのディスクにルートディレクトリを作成します。ディレクトリのロケーション(論理セクタ番号)および最大のエントリ数は、メディアによって決まります。

ルートディレクトリ以外のディレクトリは、実際にはファイルと同じなので、無制限にファイルディレクトリを作成することができます。

ディレクトリの長さは32バイトで、以下のようなフォーマットで記入されます(オフセットは16進)。

オフセット		サイズ (バイト)	内 容
16 進	10 進		
00H~07H	0 ~ 7	8	ファイル名
08H~0AH	8 ~ 10	3	拡張子
0BH	11	1	属 性
0CH~15H	12~21	10	予約エリア
16H~17H	22~23	2	最終編集時刻 ビット 0 ~ 4 = 秒/2 5 ~ 10 = 分 11 ~ 15 = 時
18H~19H	24~25	2	最終編集日付 ビット 0 ~ 4 = 日 5 ~ 8 = 月 9 ~ 15 = 年
1AH~1BH	26~27	2	開始クラスタ
1CH~1FH	28~31	4	ファイルサイズ・バイト単位

0 ~ 7 ファイル名 8 文字。8 文字に満たない場合は左詰めで、残りにスペースが入ります。このフィールドの先頭バイトは、ステータスを示します。

00H 未使用。性能上の理由から、ディレクトリ検索の長さを制限するためのものです。

05H ファイル名の先頭の 1 文字が実際には E5H であることを示します。

E5H 使用されたファイルで、すでに消去されています。

2EH このエントリは、ディレクトリのためのものです。2 バイト目も 2EH の場合、クラスタフィールドには、このディレクトリの親ディレクトリのクラスタ番号が入っています(親ディレクトリがルートディレクトリの場合、0000 H)。

これ以外の文字の場合、ファイル名の先頭の文字になります。

8 ~ 0A ファイル名拡張子

0B ファイルのアトリビュート(属性)。アトリビュートバイトは、以下のようにマップされます(値は 16 進)。

01 ファイルは、リードオンリーになっています。ファンクションコール 3DH により、このファイルを書き込みのためにオープンしようとしても、エラーコードが返されます。この値は、以下の他の値と一緒に使用することができます。ファイルの削除(13H)、ディレクトリの削除(41H)もエラーになります。

- 02 隠されたファイル。このファイルは、通常のディレクトリ検索から除外されます。
- 04 システムファイル。このファイルは、通常のディレクトリ検索から除外されます。
- 08 このエントリの最初の 11 バイトには、ボリュームラベルが入っています。このエントリは、作成の日時以外には一般的な情報が入っておらず、ルートディレクトリにのみ存在することができます。
- 10 このエントリはサブディレクトリを定義し、通常のディレクトリ検索から除外されます。
- 20 保存ビット。このビットは、ファイルがライトされクローズされたとき常に、オンにセットされます。このビットは、他のアトリビュートビットと一緒に使用することができます。

注意：IO.SYS, MSDOS.SYS には、リードオンリー、隠されて見えない、システムファイルのマークが付けられます。ファイルには作成時に、隠されて見えないファイルのマークを付けることができます。またリードオンリー、隠されたファイル、システムおよび保存の属性は、ファンクション 43 H によって変更可能です。

0C ~ 15 予約域

- 16~17 ファイルが作成された時刻または最後に編集された時刻が、つぎのようなビット列にマップされます。(左がビット 7, 右がビット 0 です)

オフセット 17H								オフセット 16H							
15				11	10					5	4				0
H	H	H	H	H	M	M	M	M	M	M	S	S	S	S	S
時					分					秒/2					

ここで、

時 2 進数で表した時刻 (0 ~ 23)

分 2 進数で表した分 (0 ~ 59)

秒 秒/2

- 18 ~ 19 ファイルが作成された時または最後に編集された日付。
年/月/日は、以下のようなビット列にマップされます。

オフセット 19H								オフセット 18H							
15						9	8			5	4				0
Y	Y	Y	Y	Y	Y	Y	M	M	M	M	D	D	D	D	D
年							月				日				

ここで、

月は 1～12

日は 1～31

年は 0～99 (1980～2079)

1A～1B 開始クラスタ。ファイルの先頭クラスタの相対クラスタ番号。

すべてのディスクのデータスペースの先頭のクラスタは、クラスタ 002 です。

クラスタ番号は、最下位バイトから先に記憶されます。

注意：クラスタ番号を論理セクタ番号に変換する場合の詳細については、3. 5. 1, 3. 5. 2 を参照してください。

1C～1F バイトで表したファイルの大きさ、最初のワードには、ファイルの大きさの下位の部分が入っています。両方のワードとも、下位のバイトから先に記憶されます。

3.5 MS-DOS ファイルアロケーションテーブル

本章は、デバイスドライバを開発するシステムプログラムのためのもので、ファイルのクラスタを、ファイルのためにディスクを割り当てる論理セクタ番号に変換するために、MS-DOS ではどのようにファイルアロケーションテーブル (FAT) が使用されるかを説明しています。ディスク上の論理セクタの位置決めは、ドライバが行います。この情報は、ドライバ以外の目的に使用すべきではありません。システムユーティリティプログラムは、FAT に直接アクセスするのではなく、MS-DOS ファイル管理ファンクションコールを使用すべきです。

FAT は、通常各クラスタごとに 12 ビットのエントリで作成されます。ただし、クラスタ数の最大値が 4085 を超えるような種類のディスクでは 16 ビットのエントリで作成されます。12 ビットのエントリの場合、先頭の 2 つの FAT のエントリはディレクトリの一部を示しており、これらの FAT にはディスクの大きさとフォーマットを示す標識が入っています。2 バイト目と 3 バイト目には、常に FFH が入っています。

3 番目の FAT から、データ領域のマッピングが始まります (クラスタ 002)。各エントリにも、16 進で表した 3 文字 (16 ビットエントリの場合は 4 文字) が入っています。

(0)000 クラスタは未使用で、使用可能です。

(F)FF7 クラスタに、スキップセクタが入っています。MS-DOS は、このようなクラスタは割り当てません。このクラスタ数が CHKDSK によって数えられ、通知されます。

(F)FF8～(F)FFF ファイル内の最終クラスタを、示しています。

(X)XXX 上記以外の 16 進数の場合ファイル内のつぎのクラスタのクラスタ番号を示しています。ファイルの先頭のクラスタ番号は、ディレクトリエントリに保存されます。

ファイルアロケーションテーブルは、常に予備セクタのつぎの最初のセクションから始まります。FAT が 1 セクタより大きい場合、これらのセクタは連続しています。ディスクには通常 FAT のコピーが信頼性を高めるために 2 つ作られています。FAT は、必要なとき（ファイルのオープン、これ以上のスペースを割り当てるなど）に MS-DOS バッファの 1 つに読み込まれます。性能上の理由から、このバッファには高いプライオリティ（順位）が与えられ、可能な限り長くメモリ内に保存されます。

3.5.1 12 ビット FAT エントリ

まずディレクトリエントリから、ファイルの開始クラスタの番号を取得します。ファイルのつぎに来るクラスタの位置を指定する場合、以下のことを行います。

1. 現在使用されているクラスタ番号に、1.5 を掛ける（各 FAT エントリは、1.5 バイトの長さです）。
2. この積全体が FAT 内のオフセットで、現在使用されているクラスタをマップするエントリをポイントしています。このエントリには、ファイル内のつぎのクラスタのクラスタ番号が入っています。
3. 計算された FAT オフセットにある 1 ワードをレジスタ内に入れるために MOV 命令を使用する。
4. 使用された最終クラスタが偶数の場合、このレジスタの内容に FFFH を加算することによってこのレジスタの下位 12 ビットを保存するか、または SHR 命令を使用してこのレジスタの内容を右に 4 ビットシフトすることによって上位 12 ビットを保存してください。
5. 結果として取得された 12 ビットが FF8H から FFFH までの値を取る場合、ファイル内にこれ以上のクラスタは存在しません。これ以外の値の場合、この 12 ビットには、ファイル内のつぎのクラスタのクラスタ番号が入っています。

このクラスタを論理セクタ番号（割り込みタイプ 25H と 26H および SYMDEB によって使用されるような、相対セクタ）に変換する場合、以下のことを行ってください。

1. クラスタ番号から 2 を引く。
2. この演算結果に 1 クラスタ当りのセクタ数を掛ける。
3. データ領域内の開始論理セクタ番号を加える。

3.5.2 16 ビット FAT エントリ

まずディレクトリエントリから、ファイルの開始クラスタの番号を取得します。
つぎのファイルのクラスタの位置を指定する場合、以下のことを行います。

1. 現在使用されているクラスタ番号に、2 を掛ける（各 FAT エントリは、2 バイトの長さです）。
2. 計算された FAT オフセットにある 1 ワードをレジスタ内に入れるために `MOV WORD` 命令を使用する。
3. 結果として取得された 16 ビットが `FFF8H` から `FFFFH` までの値を取る場合、ファイル内にこれ以上のクラスタは存在しません。これ以外の値の場合、この 16 ビットには、ファイル内のつぎのクラスタのクラスタ番号が入っています。

第4章

MS-DOSコントロールブロックとワークエリア

4.1 MS-DOS メモリマップ

XXXX:0000	割り込みのベクタテーブル
XXXX:0000	IO.SYS——MS-DOS とハードウェアのインターフェイス
XXXX:0000	MSDOS.SYS——MS-DOS 割り込みハンドラ、サービスルーチン(割り込みタイプ 21H)、MS-DOS バッファ、コントロールエリアおよび登録されているデバイスドライバ
XXXX:0000	COMMAND.COM の常駐部——割り込みタイプ 22H (終了アドレス)、23H (CTRL-C による抜け出しアドレス)、24H (致命的エラーによる打ち切りアドレス) のための割り込みハンドラおよび非常駐部分をロードし直すためのコード
XXXX:0000	外部コマンドまたはユーティリティ (.COM, .EXE ファイル)
XXXX:0000	.COM ファイルのためのユーザースタック (256 バイト)
XXXX:0000	COMMAND.COM の非常駐部——コマンドプロセッサ、内部コマンド、バッチプロセッサ

ユーザーメモリは、メモリに対するリクエストの条件を満たす、使用可能な一番低いメモリの終わりから、割り当てられます。

4.2 MS-DOS プログラムセグメント

外部コマンドを入力した場合、または EXEC ファンクションコールによってプログラムをコールした場合、MS-DOS は、使用可能な最下位のアドレスを、コマンドやプログラムのためのメモリの開始アドレスとします。この領域はプログラムセグメントと呼ばれます。

最初の 256 バイトはプログラムがロードされたときに、EXEC システムコールによってセットアップされます。プログラムはこのブロックのつぎにロードされます。EXE 形式のファイルの minalloc と maxalloc が共にゼロの場合、ファイルは可能な限り高いアドレスへロードされます。

プログラムセグメント内のオフセット 0H 以降に、プログラムセグメントプレフィクス (PSP) というコントロールブロックが MS-DOS によって作成されます (以下を参照してください)。

EXEC からプログラムを戻す場合、以下の 5 つの方法のいずれかを使用します。

1. AH=4CH で INT 21H を行う。
2. AH=31H で INT 21H を行う (KEEP PROCESS)。
3. プログラムセグメントプレフィクス内のオフセット 0 に long ジャンプを行う。
4. INT 20H を行う (CS:0 は PSP を指していなければいけません)。
5. AH=0 で INT 21H を行う (CS:0 は PSP を指していること)

注意：機能的で、将来の MS-DOS のバージョンに対応しやすいので 1 または 2 を行う方法が望ましいでしょう。

5 つの方法のいずれを使用した場合でも、結果として EXEC のコールを行ったプログラムに制御が渡されます。ただし、1 と 2 の方法は戻るときの終了コードを指定できます。戻るとき、割り込みベクタ 22H, 23H, 24H (終了アドレス、〈CTRL-C〉抜け出しアドレス、致命的エラーによる打ち切りアドレス) のアドレスが終了したプログラムのプログラムセグメントプレフィクス内に保存されていた値により回復します。こうしてつぎに制御が終了アドレスに渡されます。COMMAND に戻るプログラムの場合、制御は COMMAND の常駐部に渡され、バッチファイルを処理中の場合は、これを続行します。それ以外の場合、COMMAND によって非常駐部に対するチェックサムが行われ、必要な場合再ロードが行われます。つぎに COMMAND はシステムプロンプトを出力し、キーボードからのつぎの入力を待ちます。

プログラムに制御が渡されたとき、つぎの条件が成立します。

(1) すべてのプログラムに適用

渡された環境のセグメントアドレスは、プログラムセグメントプレフィックスのオフセット 2CH に入っています。

この環境とは、以下のフォーマットによる一連の ASCII スtrings (合計が 32K 未満) のことです。

NAME =パラメータ

各 String は、1 バイトのゼロによって区切られ、Strings 全体はさらに 1 バイトのゼロが続くことによって終了します。その最後のゼロに続くものは、ASCIZ 文字列のプログラムに 1 組のワード数を渡す引数 (初期状態) です。もし、カレントディレクトリ中でファイルが見つければ、ASCIZ 文字列は、EXEC システムコールと同じように実行可能なプログラムのドライブ名、パス名を渡します。もし、設定されたパスで、ファイルが見つければ、ファイル名はパスの情報とリンクされたものになります。プログラムは、この領域をプログラム自身がロードされた場所を知るのに使われます。コマンドプロセッサによって作成された環境 (コールを行ったすべてのプログラムに渡された) には、最低 COMSPEC=String が入っています (COMSPEC のパラメータは、ディスク上の COMMAND.COM の位置指定を行うために MS-DOS によって使用されるパスを定義します)。PATH および PROMPT コマンドもまた、MS-DOS の SET コマンドを通して入力されたすべての環境 String と一緒に環境の中に入れられます。

ユーザープログラムに渡された環境は、実際にはコールを行った環境のコピーです。ユーザーの応用プログラムで“プログラムを在駐させたまま終了”の概念を使用している場合、ユーザープログラムに渡された環境のコピーが静的であることに注意しなければなりません。すなわち、つぎに SET, PATH または PROMPT コマンドが入力された場合でも、このコピーは変更されません。逆に、元のプロセス環境で、アプリケーションによるコピーされた環境のどんな変更もできません。たとえば、SET コマンドなどで設定された MS-DOS の環境変数を変えることはできません。

プログラムセグメントプレフィックス内のオフセット 50H に MS-DOS ファンクションディスパッチャのコールを行うためのコードが入っています。したがって、指定したいファンクション番号を AH に入れ、割り込みタイプ 21H をかけるのではなく、PSP+50H に対する long コールによって MS-DOS ファンクションを行うことができます。これはコールであり、割り込みではないので、この位置にシステムコールを行うための該当するすべてのコードを入れることができます。これによってシステムのコールを行う処理を、移植性のあるものにします。

ディスク転送アドレス (DTA) は、80H にセットされます (プログラムセグメントプレフィックス内のデフォルト DTA)。

プログラムセグメントの 5CH および 6CH のファイルコントロールブロックには、コマンド

が入力されたとき先頭の2つのパラメータがセットされます。いずれかのパラメータにパス名が入っている場合、対応するFCBには有効なドライブ番号のみが入っており、ファイル名フィールドは無効になります。

81Hのフォーマットされていないパラメータエリアには、コマンド名につづいて入力されたすべての文字が入っており(区切り記号も含めて)、80Hには文字数がセットされます。コマンド行に<, >, またはパラメータが入力された場合、これら(およびこれに関連したファイル名)はこのパラメータエリアに入れられません。標準入出力の転送は、アプリケーションプログラムが意識する必要はないからです。

オフセット6(1ワード)には、セグメント内の使用可能なバイト数が入っています。

AXレジスタには、先頭の2つのパラメータ中のドライブ名が妥当かどうかを表す情報が返されます。

AL=FFH 第1のパラメータに、無効なドライブ名が入っている場合(他は、AL=00)

AH=FFH 第2のパラメータに、無効なドライブ名が入っている場合(他は、AH=00)

オフセット2(1ワード)には、利用できないメモリの先頭バイトを示すセグメントアドレスが入っています。プログラムは、アロケートメモリシステムコール(48H)が行われるまで、このアドレスを変更してはいけません。

(2) .EXE プログラムに適用

DS, ESレジスタは、プログラムセグメントプレフィクスを示すようにセットされます。

CS, IP, SS, SPレジスタは、リンカによって渡された値にセットされます。

(3) .COM プログラムに適用

4つのセグメントレジスタに、プログラムセグメントプレフィクスのコントロールブロックセグメントアドレスが入っています。

すべてのユーザーメモリが、プログラムに割り当てられます。あるプログラムがEXECファンクションコールによって他のプログラムのコールを行う場合、第2のプログラムのためのスペースを用意する目的で、セットブロック(ファンクション4AH, コード00H)ファンクションコールを通して、最初にいくらかのメモリを解放しなければなりません。

命令ポインタ(IP)は、100Hにセットされます。

SPレジスタは、プログラムセグメントの終わりにセットされます。オフセット6にあるセグメントの大きさは、この大きさのスタックを可能にするために100Hだけ縮小されます。

ゼロの入った1ワードが、このスタックのトップに入れられます。これはユーザープログラムが、RETによってCOMMANDに戻るためのものです。ただしそのために、ユーザープログラムがスタックとコードセグメントを管理することを前提としています。

プログラムセグメントプレフィクス (オフセットは 16 進) は、以下のようにフォーマットされています。

0	INT 20H	alloc. ブロック の最後 ^①	リザーブ	MS-DOS 機能をロングコールするた めの 5 バイト (オフセットアドレス) ^②
8	MS-DOS をロングコールする ためのセグメントアドレス	終了アドレス (IP, CS)		〈CTRL-C〉の抜け出 しアドレス (IP)
10	〈CTRL-C〉の抜け出 しアドレス (CS)	ハードエラーによる抜け出しアドレス (IP, CS)		
<div>16 H ~ 5 BH リザーブ (MS-DOS が使用)</div> <div>2CH^③</div>				
5 CH パラメータ 1 (通常はオープンされていない FCB)				
6 CH パラメータ 2 (通常はオープンされていない FCB, 5CH の FCB がオープンされていると、オーバーライトされる)				
80	パラメータ 3 (通常は DTA) 初期化されたコマンドインボケーションライン			
100				

注意：

- ① 使用可能なメモリ中の最初のセグメントは、セグメント (パラグラフ) のフォーマットで表します (たとえば、1000H は 64K を表します。)
- ② オフセット 6 にある 1 ワードには、セグメント内で使用可能なバイト数が入っています。
- ③ オフセット 2 CH にある 1 ワードには、環境のセグメントアドレスが入っています。

重要事項

PSP のオフセット 5CH 未満の部分は、プログラムによって変更しないでください。

第5章

EXEファイルの構造とローディング

リンカユーティリティ (Microsoft LINK) によって生成された EXE 形式のファイルは、つぎの2つの部分によって構成されています。

- ① リロケーションとコントロール情報
- ② ロードモジュール

コントロール情報、リロケート(再配置)情報は、ファイルの先頭の“ヘッダ”と呼ぶ領域に入っています。ロードモジュールはヘッダのすぐ後に位置します。ロードモジュールは、パラグラフの境界から開始し、リンカによって生成されるモジュールのメモリイメージのことです。ヘッダは、以下のようなフォーマットをしています。

オフセット(16進)	内容
00~01	4DH, 5AH——ファイルが有効な EXE 形式のファイルであることを示すため LINK プログラムによって付けられたマークです。
02~03	最後のページに入っているバイト数。オーバーレイによる読み込みに有用です。
04~05	512 バイト (ページ) 単位の、ファイルの大きさ。(ヘッダも含む)
06~07	リロケーションテーブルの項目数。この表はヘッダの直後に置かれます。
08~09	ヘッダのサイズ (16 バイトパラグラフ単位) ロードモジュールの開始点に位置指定に使用されます。
0A~0B	ロードされたプログラムの後に必要とされる 16 バイトパラグラフの最小数 (minalloc)。
0C~0D	ロードされたプログラムの後に必要とされる 16 バイトパラグラフの最大数(maxalloc)。minalloc と maxalloc が両方ともゼロのときは、プログラムはできるだけ上位にロードされます。

0E~0F	ロードモジュール内のスタックセグメントのオフセット（セグメントのフォーム）
10~11	モジュールに制御が渡されたとき、SP レジスタに返される値
12~13	ワードチェックサム——オーバーフローを無視した、ファイル内の全ワードのネガティブサム
14~15	モジュールに制御が渡されたとき、IP レジスタに返される値。
16~17	ロードモジュール内のコードセグメントのオフセット（セグメントのフォーム）
18~19	ファイル内の先頭のリロケーション項目のオフセット値。
1A~1B	オーバーレイ番号（プログラムの常駐部分は0）

上記の項目の後に、リロケーションテーブルが置かれます。このテーブルは、変数のリロケーション項目によって構成されています。項目数は、オフセット 06~07 に入っています。リロケーション項目には2つのフィールド、2バイトのオフセット値と2バイトのセグメント値が入っています。これらの2つのフィールドには、モジュールに制御が渡される前に修正を必要とする、ワードのロードモジュール中のオフセットが入っています。このプロセスは、リロケーション（再配置）と呼ばれ、つぎのように処理されます。

1. ヘッダのフォーマットが行われている部分がメモリ中に読み込まれます。ヘッダの大きさは、1BH です。
2. メモリの一部がロードモジュールサイズとアロケーションユニット数(0A~0B, 0C~0D)によってアロケートされます。まず MS-DOS はパラグラフ FFFFH をアロケートするように試みます。これは常にエラーとなりますが、結果として、最大フリーブロック数が返されます。もしこのブロック数が minalloc とロードサイズよりも小さい場合は、ノーメモリエラーとなります。また、もしこのブロック数が maxalloc とロードサイズよりも大きいならば、MS-DOS はアロケートを行います(maxalloc+ロードサイズ)。さもなければ、MS-DOS はメモリの最大フリーブロックにアロケートを行います。
3. プログラムセグメントプレフィクスが、アロケートされたメモリの最低位に作られます。
4. ロードモジュールの大きさは、ファイルの大きさ（オフセット 04H~05H）からヘッダの大きさ（08H~09H）を引くことによって決定されます。実際の大きさはオフセット 02~03 の内容に基づき、調整が行われます。Microsoft LINK の high/low スイッチのセッティング

に基づき、ロードモジュールをロードするための該当するセグメントが決定されます。このセグメントは、スタート（開始）セグメントと呼ばれます。

5. ロードモジュールが、スタートセグメントからメモリへロードされます。
6. リロケーションテーブル項目は、ワークエリアに読み込まれます。
7. 各リロケーションテーブル項目のセグメント値が、スタートセグメント値に加算されます。この計算されたセグメントはリロケーション項目オフセット値と共に、ロードモジュール内のワードを示します。演算結果は、ロードモジュール内のワードに返されます。
8. いったんすべてのリロケーション項目が処理されると、SS, SP レジスタは、ヘッダ内の値によりセットされ、スタートセグメント値が SS に加算されます。ES, DS レジスタは、プログラムセグメントプレフィクス内のセグメントアドレスにセットされます。スタートセグメント値が、ヘッダ CS レジスタの値に加算され、この演算結果はヘッダ IP 値と共に、CS:IP の初期値としてこのモジュールに制御を渡すために使用されます。

第6章

インテルオブジェクトモジュールフォーマット

6.1 イントロダクション

本章では、8086 マイクロプロセッサ(8086 と上位互換性のあるものを含む：以降、単に 8086 と呼びます) のリロケータブル (再配置可能) なオブジェクト言語を定義するオブジェクトレコードのフォーマットについて解説します。8086 オブジェクト言語は、8086 をターゲットプロセッサとし、マイクロソフト LINK でリンク (連結) 可能な、すべての言語トランスレータの出力を指します (本章の解説では、アセンブラ、コンパイラを総称し、トランスレータと呼びます)。8086 オブジェクト言語はリンカやライブラリマネージャ等のオブジェクト言語プロセッサの入力であると同時に出力でもあります。

8086 オブジェクトモジュールのフォーマットを使うと相互に連結可能であるようなりロケータブル (再配置可能) なメモリイメージを指定することができます。このフォーマットは、8086 マイクロプロセッサのメモリマップ機能を有効に使用できるように設計されています。

つぎの表は、マイクロソフト社が採用しているレコードフォーマットの一覧です。このレコードフォーマットについては本章中で説明しております。表中で、前にアスタリスク (*) マークがつけられたレコードフォーマットは、インテル仕様に準じたものであることを示します。

表6.1 オブジェクトモジュールのレコードフォーマット

T-モジュール ヘッダレコード
ネームリストレコード
*セグメント定義レコード
*グループ定義レコード
*タイプ定義レコード
シンボル定義レコード
*パブリック名定義レコード
*エクスターナル名定義レコード
*行番号レコード
データレコード
論理データレコード (繰り返し参照されない)
論理データレコード (繰り返し参照される)
FIXUP レコード
*モジュールエンドレコード
コメントレコード

6.2 用語の定義

8086 の再配置とリンクの基礎となる用語を、以下に示します。

OMF

オブジェクトモジュールフォーマット (Object Module Format)。

MAS

メモリアドレス空間 (Memory Address Space)。

8086MAS は 1M (メガ: 1048576) バイトです。

この MAS は、実メモリ (MAS の一部分になる) とは区別されることに注意してください。

MODULE (モジュール)

トランスレータによって生成したオブジェクトコードと、他の情報の分割不可能な集合

T-MODULE (T-モジュール)

PASCAL や FORTRAN のようにコンパイラ／アセンブラが生成したモジュール

オブジェクトモジュールはつぎの制限を受けます。

1. 各モジュールには名前がつけられます。トランスレータは T-モジュールに名前を与えますが、ソースコードも使用者も他の名前を指定しない時、デフォルト名(通常ファイル名、または空名称)が使われます。
2. シンボリックデバッガが各種の行番号やローカルなシンボルを読み分けることができるように、リンクされたモジュールの集合体の中の各 T-モジュールはそれぞれ別の名前がつけられます。このような制限はリンカが要求するものではなく、また強制するものでもありません。

FRAME (フレーム)

パラグラフ境界 (16 の整数倍のアドレス) で始まる、MAS で 64K の連続域。8086 の 4 つのセグメントレジスタの内容が 4 つの (重なりあっても良い) フレームを定義するためにこのような 8086 コードの 16 ビットアドレスでは、その時点での 4 つのフレーム以外のメモリロケーションをアクセスすることはできません。

LSEG (論理セグメント)

コンパイル／アセンブル時(アドレス拘束時以外)に内容を決定されるメモリの連続域。MAS 中のサイズおよびロケーションをコンパイル時に決定する必要はありません。リンク時に LSEG は他の LSEG と結合して 1 つの LSEG を形成しますので、サイズは、各 LSEG 内で部分的に固定されているけれども最終的なものではありません。LSEG は、フレーム内に収まらなければならないので、サイズは 64K バイト以内です。LSEG のどの領域も、その LSEG を含むフレームのベースから 16 ビットオフセットだけでアドレス指定することができます。

PSEG (物理セグメント)

この語はフレームと同一です。「PSEG」と「LSEG」は注目しているセグメントの「物理的」、「論理的」の区別を表しているので、場合によってはこの語が選んで使われることもあります。

FRAME NUMBER (フレーム番号)

各フレームはパラグラフ境界から始まります。MAS の「パラグラフ」は 0 から 65535 までの番号をつけることができます。この番号はそれぞれフレームを定義するのでフレーム番号と呼ばれます。

PARAGRAPH NUMBER (パラグラフ番号)

フレーム番号と同一です。

PSEG NUMBER (PSEG 番号)

フレーム番号と同一です。

GROUP (グループ)

翻訳 (コンパイル/アセンブル) 中に決まる LSEG の集合のこと。その集合の MAS 中における最終的な位置については、その集合中の各 LSEG をカバーできるフレームが少なくとも 1 つは存在しなくてはならないという制約を受けています。

「Gr A (X, Y, Z)」は LSEG である X, Y, Z が A という名前のグループを形成することを示しています。X, Y, Z が同じグループに含まれる LSEG であるという事実は、MAS 中の X, Y, Z の順番や、X, Y, Z 間の連続性を表すものではありません。

現在、マイクロソフト LINK では LSEG を複数のグループに属させることはできません。リンクは複数のグループへの LSEG の位置づけを無視します。

CANONIC (正規)

MAS 中のロケーション (アドレス) に注目して見ると、それを含むフレームは 4096 通り考えることができます。

この 4096 通りのフレームの中のフレーム番号の最大のものだけを区別し、特別にそのロケーションの正規フレームと呼びます (あるバイトの正規フレームとは、そのフレームからのバイトオフセットが 0~15 の範囲に入るように選択されたフレームということです)。したがって FOO がメモリロケーションを定義したシンボルである時には、「FOO の正規フレーム」というように使うことができます。

拡張すると (S を何かメモリロケーションの集合としたとき)、S 中のロケーションについての正規フレームの集合中で最下位のフレーム番号を持つフレームは、ただ 1 つ存在します。この特定のフレームを集合 S の正規フレームと呼びます。よって、LSEG の正規フレームとか、LSEG のグループの正規フレームとか呼ぶことができます。

SEGMENT NAME (セグメント名)

LSEG は翻訳 (コンパイル/アセンブル) 時に、セグメント名を割り当てられます。この名前の割り当ては、つぎの目的で行われます。

1. リンク時にどの LSEG が他の LSEG と連結されるのかを決める役割を果たします。
2. グループを指定するために、アセンブラリソースコード中で使用されます。

CLASS NAME (クラス名)

LSEG には、翻訳時に、オプションでクラス名を割り当てることができます。同じクラス名

を持つ2つのLSEGは同一クラスに属していることになります。

マイクロソフトLINKはつぎの意味で名前をクラス付けします。「CODE」というクラス名や、語尾に「CODE」を含むクラス名は、そのクラスがコードのみを含んでおり、読み出すことしかできないことを意味します。そのようなセグメントである場合には、オーバーレイの一部として、そのセグメントを含むモジュールを指定すれば、オーバーレイすることができます。

OVERLAY NAME (オーバーレイ名)

LSEGには、オプションとしてオーバーレイ名を割り当てることができます。マイクロソフトLINK (V2.40以降)はLSEGオーバーレイ名を無視しますが、インテルの再配置(relocation)と連結(リンク:linkage)のツールではこれを使用することができます。

COMPLETE NAME (コンプリート名)

LSEGのコンプリート名はセグメント名、クラス名、オーバーレイ名で構成されます。別々のモジュール中のLSEGは、そのコンプリート名が同一であればリンク(連結)されます。

6.3 モジュールの一致と属性

モジュールのヘッダレコードは、モジュール中で常に最初のレコードとなり、これはモジュール名を与えます。

名前に加えて、モジュールは、指定された開始アドレスを有するものと同様に、主プログラムとしての属性をもつことができます。複数のモジュールを連結する時は、主プログラムの属性を持つモジュールを1つだけ与えます。

これは、モジュールは主(プログラム)となる場合と、ならない場合があり、また開始アドレスを持つ場合と持たない場合があることを示します。

6.4 セグメント定義

モジュールは、トランスレータによって生成されるレコードの並びによって定義されるオブジェクトコードの集まりであるといえます。オブジェクトコードは、翻訳(コンパイル/アセンブル)時に内容を決定されるメモリの連続域を表現しています。この領域を論理セグメント(LSEG)と呼びます。

モジュールは、各LSEGの属性を定義します。セグメント定義レコード(SEGDEF)はすべてのLSEG情報(名前、レコード長、メモリ配置等)を維持する媒体です。複数のLSEGがリンクされていて、セグメントアドレス可能性(6.5の「セグメントアドレッシング」を参照してください)が確立されている時は、LSEG情報が必要になります。SEGDEFレコードは、最初のヘッダレコードの後に置かれなければなりません。

6.5 セグメントアドレッシング

8086 アドレス指定の機構は、64K バイトのメモリ領域（フレームと呼ばれる）をアドレッシングするためのセグメントベースレジスタを用意しています。これらには1つのコードセグメントベースレジスタ（CS）と2つのデータセグメントベースレジスタ（DS、ES）、1つのスタックセグメントベースレジスタ（SS）があります。

メモリイメージを作り上げる LSEG の数の最大値は使用可能なベースレジスタの数をはるかに上まわります。よってベースレジスタはその度にロードしてやる必要があります。たとえば小さなデータ LSEG や、レコード LSEG がたくさん集まって作られたモジュールプログラムの場合です。

ベースレジスタをその度にロードするのはあまり望ましくないため、1つのメモリフレームに納まる単一ユニットに、多くの小さい LSEG を集め、同じベースレジスタ値を使用してすべての LSEG をアドレッシングできるようにするのがよいでしょう。このアドレッシング可能なユニットはグループといい、6.2 の“用語の定義”で定義されています。

グループ中のオブジェクトのアドレッシングの可能性を確立するためには、グループはモジュール中で明確に定義されていなくてはなりません。グループ定義レコード（GRPDEF）は、セグメント名や、「シンボル FOO を定義するセグメント」または「ROM というクラス名を持つセグメント」のような属性などによる、構成セグメントのリストを与えます。

モジュール中の GRPDEF レコードは、すべての SEGDEF レコードの後に置かれなくてはなりません。これはグループを定義するのに GRPDEF レコードが SEGDEF レコードを参照するからです。また、GRPDEF レコードは、リンカが最初に処理しなくてはならないため、他のすべてのレコード（ヘッダレコードを除く）より先に置かれなければなりません。

6.6 シンボル定義

マイクロソフト LINK にはシンボル定義レコードのクラスになる、3種類のレコードを採用しています。そのうち2つは重要で、それらはパブリック名定義レコード（PUBDEF）とエクスターナル名定義レコード（EXTDEF）です。これらはグローバルに参照可能なプロシージャとデータ項目を定義し、外部参照を解決するのに使われます。さらに TYPDEF レコードはマイクロソフト LINK が共有変数の割り当てをするのに使われます。6.14 の“共有変数の型に関するマイクロソフト表現法”を参照してください。

6.7 インデックス

「インデックス」フィールドは本書中あらゆる所に出てきます。インデックスは数値の項目の集合中から特定のものを選択する整数です。たとえば、名前インデックス、グループインデックス、エクスターナルインデックス、型インデックスなどがあります。

注意：

インデックスは通常正の数です。インデックス値の0は予約されており、インデックスの型によって特別な意味を持たせることもあります。(つまり、セグメントインデックスが0の時は「名前なし」の擬セグメントであることを示し、また型インデックスが0の時は、「型なし」のセグメントで、「指定なし」とは区別されることを示す等です。)

一般的に、インデックスは値が非常に大きいことまでを想定しています(つまり、255をはるかに超えること)。にもかかわらず、オブジェクトファイルの多くは、50や100を超えるインデックスを含みません。したがって、必要に応じて1~2バイトでインデックスはコード化されます。

第1バイト(おそらくはこれのみ)の高位(最も左の)ビットはインデックスが1バイトを占めるか2バイトを占めるかを決定します。そのビットが0である場合はインデックスは0~127になり、1バイトを占めます。そのビットが1の場合は、インデックスは0~32767の値を取り、2つのバイトは、下位8ビットが第2バイト、上位7ビットが第1バイトとなります。

6.8 フィックスアップのためのフレームの概念

「フィックスアップ」はオブジェクトコードに与えるある変更であり、これはトランスレータによって要求され、リンカによって実行され、アドレスの結合を達成します。

注意：

前述の「フィックスアップ」の定義は正確にはリンカの側からの視点を表します。にもかかわらず、リンカはこの定義に合わないオブジェクトコードの変更(すなわち、「フィックスアップ」)を行うのに使われることもあります。たとえば(オブジェクト)コードをハードウェア浮動小数点、またはソフトウェア浮動小数点サブルーチンに連結することはオペレーションコードへの変更になります(この時オペレーションコードはアドレスとして取り扱われている必要があります)。前出の「フィックスアップ」の定義はオブジェクトコードの変更を禁じるものでも軽んじるものでもありません。

8086のトランスレータはつぎの4つのデータを与えることによりフィックスアップを指定します。

1. フィックスアップする。ロケーションの場所と型
2. 2つあるフィックスアップ。MODE（モード）のうちのどちらか。
3. ターゲット。ロケーションが参照しなくてはならないメモリアドレス。
4. 参照をした文脈を定義するフレーム。

LOCATION（ロケーション）

ロケーションには5種類ありますが、それはポインタ、ベース、オフセット、HIBYTE（高位バイト）、LOWBYTE（低位バイト）です。

つぎの図6.1の縦のアライメントは4つの点を示します（8086メモリの1ワード中の高位バイトとは高位のアドレスを持つバイトであることに注意してください）。

1. ベースはポインタ中の高位ワードです（リンカはポインタの低位ワードが存在するか否かには関与しません）。
2. オフセットはポインタの低位ワードです（また、リンカは高位ワードが続くか否かには関与しません）。
3. HIBYTEはオフセットの高位側の半分です（リンカは、低位側の半分が前にあったか否かには関与しません）。
4. LOBYTEはオフセットの低位側の半分です（リンカは高位側の半分が存在するか否かには関与しません）。

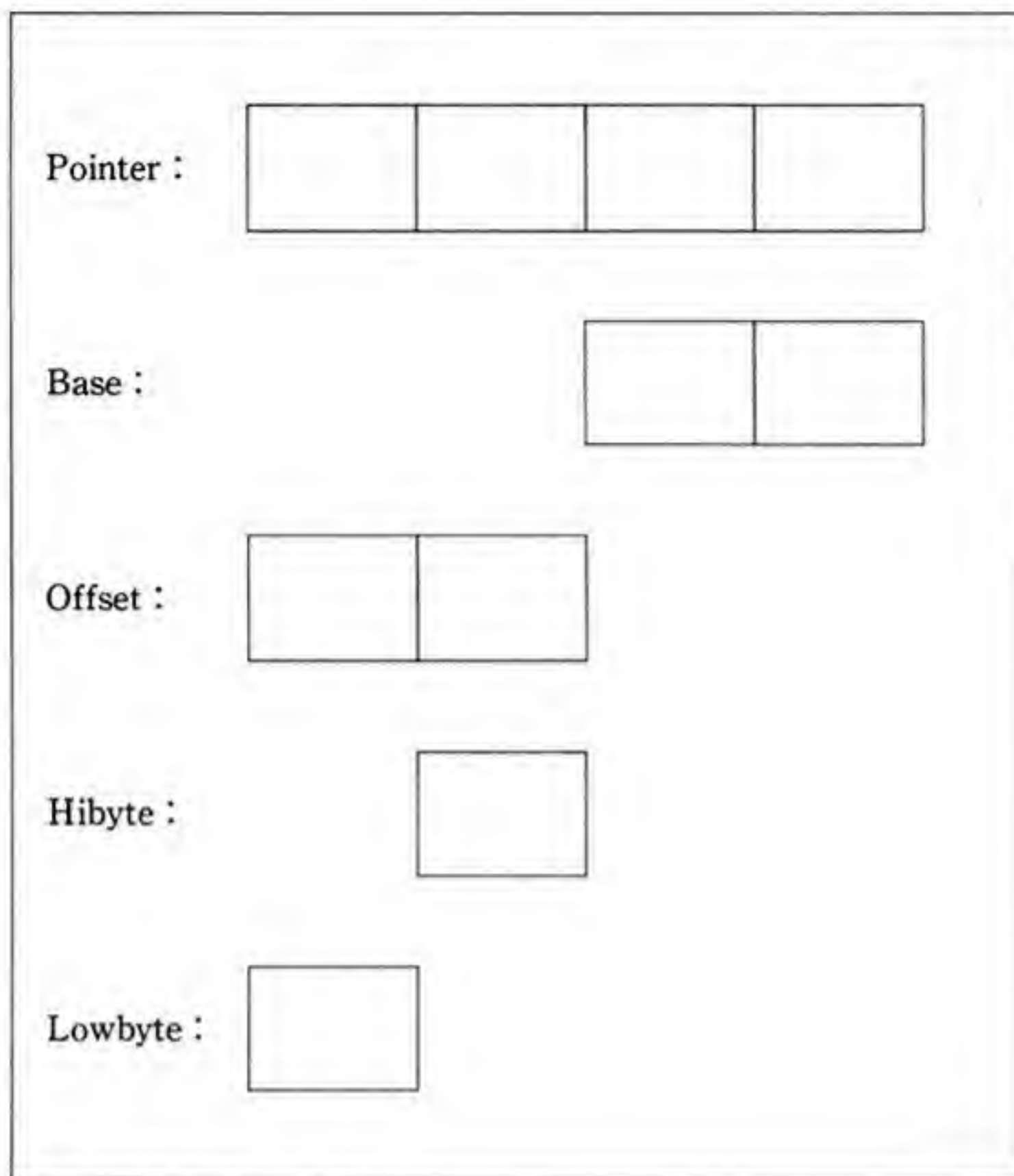


図 6.1 ロケーションのタイプ

ロケーションは2つのデータによって指定されます。(1)ロケーションの型と(2)ロケーションの場所です。(1)はフィックスアップレコード中のLOCATフィールドのLOCサブフィールドによって指定します。(2)はフィックスアップレコード中のLOCATフィールドのDATA RECORD OFFSETサブフィールドで指定します。

MODE

リンカは2種類のフィックスアップである「セルフリラティブ (自己相対)」と「セグメントリラティブ (セグメント相対)」をサポートします。

自己相対フィックスアップはCALL, JUMP, SHORT-JUMP 命令に使う8ビットと16ビットオフセットをサポートします。セグメント相対フィックスアップは、他のすべての8086アドレッシングモードをサポートします。

TARGET

ターゲットはMAS中の参照されるロケーションです(正確にいうと、ターゲットは参照されるオブジェクトの最下位バイトです)。ターゲットは、つぎの8つの方法のうちの1つで指定さ

れます。そのうち4つは「基本的」な方法であり、他の4つは「二次的」な方法です。ターゲットを指定する基本的な方法では、インデックス、またはフレーム番号 X と変位 D の2種類のデータを使用します。

- T0 X はセグメントインデックスです。ターゲットはインデックスによって識別される LSEG の D 番目のバイトです。
- T1 X はグループインデックスです。ターゲットはインデックスによって識別される LSEG の D 番目のバイトです。
- T2 X はエクスターナルインデックスです。ターゲットは、インデックスによって識別されるエクスターナル名によって（結果的に）アドレスが与えるバイトの後の D 番目のバイトです。
- T3 X はフレーム番号です。フレーム番号によって識別されるフレーム中の D 番目のバイトです（つまりターゲットのアドレスは $(X * 16) + D$ のようになります）。

ターゲットを指定する「2 次的」方法はどちらもデータ項目は1つしかとりません。インデックス、またはフレーム番号（インデックス、またはフレーム番号 X）です。暗黙のうちに変位は 0 であると仮定します。

- T4 X はセグメントインデックスです。ターゲットはインデックスにより識別される LSEG の 0 番目（最初の）バイトです。
- T5 X はグループインデックスです。ターゲットは、MAS 中で結果的に最下位に位置づけられる指定グループ中の LSEG の 0 番目（最初の）バイトです。
- T6 X がエクスターナルインデックスです。ターゲットはインデックスによって識別されるエクスターナル名のアドレスとなるバイトです。
- T7 X はフレーム番号です。ターゲットは 20 ビットアドレスが $(X * 16)$ となるバイトです。

注意：

マイクロソフト LINK では前述のうち T3 と T7 の方法は使えません。

ターゲットを記述する時はつぎのような表記法を用います。

TARGET : SI (〈セグメント名〉), 〈変位〉	[T0]
TARGET : GI (〈グループ名〉), 〈変位〉	[T1]
TARGET : EI (〈シンボル名〉), 〈変位〉	[T2]
TARGET : SI (〈セグメント名〉)	[T4]
TARGET : GI (〈グループ名〉)	[T5]
TARGET : EI (〈シンボル名〉)	[T6]

これらの表記の例をつぎに示します。

TARGET : SI (CODE), 1024

セグメント「CODE」中の 1025 番目のバイト。

TARGET : GI (DATAAREA)

MAS 中の「DATAAREA」という名前のグループのロケーション

TARGET : EI (SIN)

外部サブルーチン「SIN」のアドレス

TARGET : EI (PAYSCHEDULE), 24

「PAYSCHEDULE」という名称の外部データ構造のつぎの、24 番目のバイト

FRAME (フレーム)

各 8086 メモリ参照はいずれかのフレームに含まれるロケーションに向けられます。またフレームはいずれかのセグメントレジスタの内容によって指定されます。リンカにとって正確で、かつ使用可能なメモリ参照を行うためには、何がターゲットであり、参照すべきフレームがどこにあるかを与えなくてはなりません。このように、各フィックスアップはしかるべきフレームを、6 とおりの方法のうちの 1 つによって指定します。方法によっては前述のように、インデックス、またはフレーム番号中のデータ X を使うものがあります。これ以外はデータを必要としません。

フレームを指定する 6 つの方法をつぎに示します。

- F0 X はセグメントインデックスです。フレームはインデックスによって定義される LSEG の正規フレームです。
- F1 X はグループインデックスです。フレームはグループによって定義される正規フレームです。(つまりグループ中で最終的に MAS 中で最下位に位置づけされた LSEG によって定義される正規フレーム)。
- F2 X はエクスターナルインデックスです。フレームはエクスターナル名のパブリック定義がなされると決定されます。この時 3 つに場合わけをすることができます。
 - F2a シンボルをある LSEG に相対的に定義し、相互に関連するグループがない場合、LSEG の正規フレームが指定されます。
 - F2b シンボルは LSEG を参照することなしに絶対的に定義され、相互に関連するグループがない場合、フレームは、シンボルを定義する PUBDEF フィールドのサブフィールドであるフレーム番号によって指定されます。

- F2c シンボルの定義方法に無関係で、相互に関連するグループが存在する場合、グループの正規フレームによって指定されます。グループは、PUBDEF のサブフィールドであるグループインデックスによって指定されます。
- F3 X はフレーム番号です。これは明確にフレームを指定します。
- F4 X がない場合、フレームはロケーションを含む LSEG の正規フレームです。
- F5 X がない場合、フレームはターゲットによって決定されますが、4 つの場合に分けられます。
- F5a ターゲットがセグメントインデックスを指定する場合、この場合、フレームは (F0) と同様に決定されます。
 - F5b ターゲットがグループインデックスを指定する場合、この場合、フレームは (F1) と同様に決定されます。
 - F5c ターゲットがエクスターナルインデックスを指定する場合、この場合、フレームは (F2) と同様に決定されます。
 - F5d ターゲットが明示フレーム番号を指定する場合、この場合、フレームは (F3) と同様に決定されます。

注意：

マイクロソフト LINK ではフレーム指定法のうち F2b, F3, F5d は使えません。

フレームを記述する時もターゲットの記述と同様に行います。

FRAME : SI (<セグメント名>)	[F0]
FRAME : GI (<グループ名>)	[F1]
FRAME : EI (<シンボル名>)	[F2]
FRAME : LOCATION	[F4]
FRAME : TARGET	[F5]
FRAME : NONE	[F6]

8086 メモリ参照では、自己相対参照によって指定されるフレームは通常ロケーションを含む LSEG の正規フレームであり、セグメント相対参照によって指定されるフレームはターゲットを含む LSEG の正規フレームです。

6.9 セルフリラティブフィックスアップ

セルフリラティブ (自己相対) フィックスアップはつぎのように動作します。

メモリアドレスはロケーションによって暗黙の内に定義されます。つまり、ロケーションに

続くバイトのアドレスにより定義されます(自己相対参照時に、8086のIP(インストラクションポインタ)は参照に続くバイトをポイントしているからです)。

8086の自己相対参照の際、ロケーション、またはターゲットが指定フレームの外にある場合、リンカは警告を出します。その他の場合、ロケーションが暗黙に定義するアドレスに加えられ、一義の16ビット変位が存在します。フレーム中のターゲットの相対位置を与えることとなります。

ロケーションがオフセットの場合、変位はロケーションに加えられ、65536で割った余りが取られます。この場合、エラー発生にはなりません。

ロケーションがLOYTEの場合は、変位は-128~127の範囲でなければなりません。それ以外の場合はリンカが警告を発します。変位はロケーションに加えられ、255で割った余りが取られます。

ロケーションがベース、ポインタ、またはHIBYTEである場合、トランスレータ中で何が行われるのかが、明確に表されてなく、リンカの行う動作も定義されていません。

6.10 セグメントリラティブフィックスアップ

セグメント相対フィックスアップの動作はつぎのように行われます。負でない16ビット数、FBVALは、フィックスアップが指定するフレームのフレーム番号として定義されます。さらに符号付き20ビット数、FOVALはフレームのベースからターゲットまでの距離として定義されます。この符号付きの20ビット数が0より小さいか、または65535より大きい時、リンカはエラーを表示します。それ以外の場合は、FBVAL、FOVALはつぎのようにロケーションをフィックスアップするのに使われます。

1. ロケーションがポインタである場合、FBVALは(MOD 65536: MODは剰余計算)で、ポインタの高位ワードに加えられ、FOVALは(MOD 65536)で、ポインタの低位ワードに加えられます。
2. ロケーションがベースの場合、FBVALは(MOD 65536で)BASEに加えられますが、FOVALは無視されます。
3. ロケーションがオフセットである場合、FOVALは(MOD 65536)でオフセットに加えられますが、FBVALは無視されます。
4. ロケーションがHIBYTEの場合、(FOVAL/256)は(MOD 256で)HIBYTEに加えられますがFBVALは無視されます。(前述の除算は「整数除算」であり、余りは捨てられます。
5. ロケーションがLOBYTEの場合、(FOVALを256で割った余り)は(MOD 256で)LOBYTEに加えられます。FBVALは無視されます。

6.11 レコードオーダ

オブジェクトコードファイルは、(単数、または複数の)モジュールの連続したものを含むか、(0 か、またはそれ以上の) モジュールを含むライブラリを含む必要があります。1つのモジュールは、オブジェクトコードの集合として定義され、コードはオブジェクトレコードの連続として定義されます。つぎの構文はモジュールを形成するための、レコードの正当な階層を示します。さらに与えられた構文規則はレコード列の解決の方法に関する情報を与えます。

注意：

つぎに使う構文記述言語は WIRTH によって定義されています(CACM, 1977 年 11 月作成, ボリューム# 20, 番号# 11, # 822-# 832 ページ. 大文字で書かれているのはリテラルではなく, レコードフォーマットの説明中で定義される識別子です)。

```

object file      = tmodule
tmodule          = THEADR seg-grp {component}      modtail
seg-grp          = {LNAMES} {SEGDEF} {TYPDEF | EXTDEF | GRPDEF}
component        = data | debug-record
data             = content-def | thread-def | TYPDEF | PUBDEF | EXTDEF
debug-record     = LINNUM
content-def      = data-record {FIXUPP}
thread-def       = FIXUPP (containing only thread fields)
data-record      = LIDATA | LEDATA
modtail          = MODEND

```

つぎの規則が適用されます。

1. FIXUPP レコードは常に前の DATA (データ) レコードを参照します。
2. すべての LNAME, SEGDEF, GRPDEF, TYPEDEF, EXTDEF のレコードはこれを参照するレコードより前に与えられていなくてはなりません。
3. COMENT レコードは、ファイル中のどこにも存在できますが、ファイルやモジュール中の最初、または最後のレコードとしたり、条件レコード中には置けません。

6.12 レコードフォーマットについて

つぎにレコードフォーマットダイアグラムの概略図を示します。これはレコードフォーマットのサンプルであり、各種の規則を表したものです。

6.12.1 SAMPLE RECORD FORMAT (レコードフォーマットの例) (SAMREC)

REC XYP xxH (1)	RECORD LENGTH (2)	NAME (1以上)	NUMBER (4)	CHK SUM (1)
RPT				

タイトルと公式略称

先頭には、図示したレコードフォーマットの名前と、その公式な略称が記述されています。トランスレータおよびデバッガのような種々のプログラム間で一義性を促進するために、コードとドキュメンテーションの双方でこの略称を使うべきです。レコードフォーマットの略称は、常に6文字で示されます。

ボックス

フォーマットはボックスによって記述されます。()内の数字はそのフィールドのサイズ(バイト単位)です。

RECTYP (レコードの型)

各レコードの第1バイトは、0~255の値を取り、レコードがどの型(RECORD type)であるかを示しています。

RECORD LENGTH (レコード長)

各レコードの第2フィールドは、レコードのバイト数(初めの2つのフィールドを除く)を含みます。

NAME (名前)

「NAME (名前)」と書かれたフィールドはどれもつぎの内部構造を持ちます。1バイト目はフィールド中の残りのバイト数を示します。残りのバイトはバイトごとの文字列として翻訳(コンパイル/アセンブル)されます。

ほとんどのトランスレータは ASCII 文字セットの部分集合であるように限定しています。

NUMBER (番号)

4 バイトの NUMBER フィールドは符号なしの 32 ビット整数を示し、先頭の 8 ビット(最小有効桁)を第 1 バイト(最低位アドレス)に、続く 8 ビットを第 2 バイトに、という形で格納されています。

REPEATED OR CONDITIONAL FIELDS (反復または条件フィールド)

レコードフォーマットの一部には、数回反復されるフィールド列が含まれています。この部分は「RPT (反復)」というブラケットがボックスの下部に示されます。

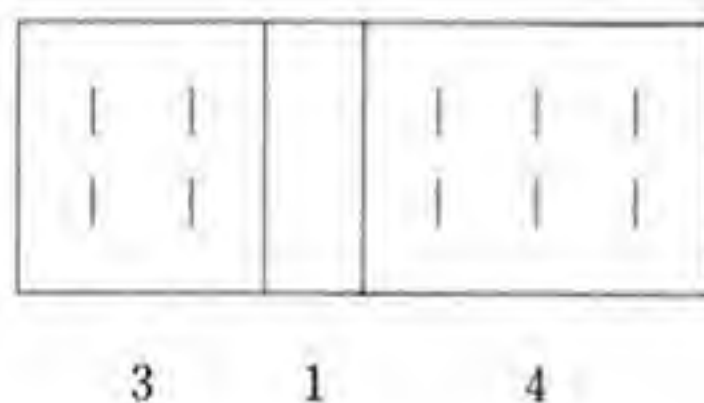
同様に、与えられた条件が正であるか否かだけを示す部分もありますが、これは同じように「COND (条件)」というブラケットがボックスの下部に示されます。

CHKSUM (チェックサム)

各レコードの最後のフィールドはチェックサムです。これはレコード中の他のすべてのバイトの合計を 2 の補数 (MOD 256) で表したものになっています。よってレコードに含まれるバイトの合計 (MOD 256 で) は 0 になります。

BIT FIELDS (ビットフィールド)

フィールド内容の記述はビットレベルであることもあります。ボックス内に縦線 (|) の引かれたボックスは、バイト、またはワードを示します。この縦線はビットの境界を意味し、つぎに示す図では、3 ビット、1 ビット、4 ビットの 3 つのビットフィールドがあることを示します。



6.12.2 T-MODULE HEADER RECORD (T-モジュールヘッダレコード) (THEADR)

REC TYP	RECORD LENGTH	T- MODULE NAME	CHK SUM
80H (1)	(2)	(1以上)	(1)

トランスレータから出力される各モジュールは T-モジュールヘッダレコードを持ちます。

T-MODULE NAME (T-モジュール名)

T-MODULE NAME は T-モジュールの名前です。

6.12.3 LIST OF NAMES RECORD (名前リストレコード) (LNAMES)

REC TYP	RECORD LENGTH	NAME	CHK SUM
96H (1)	(2)	(1以上)	(1)

RPT

このレコードは、続く SEGDEF や GRPDEF レコード中でセグメント名、クラス名や、またはグループ名として使われる名前のリストです。

モジュール中の LNAMES レコードの順序と、LNAMES レコード間での名前の順序は名前の順序を付けることになります。したがって、それらの名前に 1, 2, 3, 4……と番号を割り当てることができます。この番号はセグメント名やインデックス、クラス名インデックス、SEGDEF や GRPDEF レコードのグループ名インデックスフィールド中で「名前インデックス」として使われます。

NAME (名前)

この反復フィールドは、名前を示し、フィールド長が 0 をとることが可能です。

6.12.4 SEGMENT DEFINITION RECORD (セグメント定義レコード) (SEGDEF)

REC TYP 98H	RECORD LENGTH	SEG ATTR	SEG MENT LENGTH	SEG MENT NAME INDEX	CLASS NAME INDEX	OVER LAY NAME INDEX	CHK SUM
(1)	(2)	(1 以上)	(2)	(1 以上)	(1 以上)	(1 以上)	(1)

特定の LSEG を参照するために他のレコード型で使われるセグメントインデックス (セグメントインデックス) 値 (1~32767) は、オブジェクトファイル中に現れる SEGDEF レコード中で (列として) 暗黙の内に定義されます。

SEG ATTR (セグメントのアトリビュート)

SEG ATTR フィールドはセグメントの属性に関する情報を与え、つぎのフォーマットで示します。

ACBP	FRAME NUMBER	OFF SET
(1)	(2)	(2)
COND		

ここに ACBP バイトは属性を記述する 4 つの要素 A, C, B, P からなります。このバイトのフォーマットをつぎに示します。

A	C	B	P

「A」(Alignment: アライメント (配置, 配列)) は LSEG のアライメント属性を指定する 3 ビットのサブフィールドです。つぎにその意味を示します。

- A=0 SEGDEF は絶対 LSEG を定義します。
- A=1 SEGDEF はリロケートブルなバイトアライメントの LSEG を定義します。
- A=2 SEGDEF はリロケートブルなワードアライメントの LSEG を定義します。
- A=3 SEGDEF はリロケートブルなパラグラフアライメントの LSEG を定義します。
- A=4 SEGDEF はリロケートブルなページアライメントの LSEG を定義します。

A=0 の場合、フレーム番号フィールドと OFFSET フィールドが存在します。マイクロソフト LINK では、アドレス指定の目的のみに使用されます。たとえば ROM の開始アドレスを定義し、ROM 内にシンボル名を定義する等です。マイクロソフト LINK は絶対 LSEG に属するデータ指定はすべて無視します。

「C」(Combination: 結合タイプ) は結合タイプを指定する 3 ビットのサブフィールドです。絶対セグメント (A=0) は結合タイプ 0 (C=0) を持ちます。リロケートブルなセグメントでは、C フィールドはセグメントがどのように組み合わせできるかを示す数 (0, 1, 2, 4, 5, 6, 7) をコードとして使用します。この (結合タイプ) 属性の解説は、2 つの LSEG の結合状態を考えるとよく理解できるでしょう。X, Y を LSEG, Z を X, Y 結合タイプの結果 (与えられる LSEG) と考えます。LX, LY を X, Y の長さ、MXY を LX, LY のうち大きい方とします。G は Y のアライメント属性に適合した、Z 中の X, Y 要素間のギャップとします。LZ は (結合している) LSEG Z の長さ、dx ($0 \leq dx < LX$) は X のオフセット (バイト単位)、同様に dy は (バイト単位の) Y のオフセットとします。つぎの表は結合している LSEG Z の長さ LZ, X 中の dx, Y 中の dy に対応する (Z に含まれる) オフセットである dx', dy' を表しています。インテルは、さらにアライメントタイプ 5 と 6 を定義し、そのアライメントタイプのセグメントのコードとデータを処理します。

表 6.2 Combination 属性の例

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	"Public"
5	LX+LY+G	dx	dy+LX+G	"Stack"
6	MXY	dx	dy	"Common"

表 6.2 を見ると C=0, C=1, C=3, C=4, C=7 に対応する行がありません。C=0 はリロケータブルな LSEG が結合されていない可能性があります。C=1, C=3 は定義されません。C=4, C=7 は C=2 と同様に取り扱われます。C1, C4, C7 はインテル規格ではすべて異なる意味を持ちます。

「B」(Big) は 1 ビットサブフィールドでこれが 1 を取る時は、セグメント長がちょうど 64K (65536) であることを示します。この場合、SEGMENT LENGTH フィールドは 0 になっていなくてはなりません。

「P」フィールドは常に 0 である必要があります。「P」フィールドはインテル仕様である「ページ常駐」フィールドです。

フレーム番号と OFFSET フィールド (絶対セグメント A=0 の時のみ存在) は絶対セグメントの MAS 中の位置づけを指定します。OFFSET の範囲は 0~15 に限られます。15 以上の値を OFFSET に与えたい時は、フレーム番号を調整する必要があります。

SEGMENT LENGTH (セグメント長)

SEGMENT LENGTH フィールドはセグメント長をバイト単位で与えます。長さは 0 であってもよいのですが、その場合でもマイクロソフト LINK はモジュールからセグメントを削除しません。セグメント長フィールドはちょうど 0~65535 を格納できる大きさを持っています。セグメントにちょうど 64K の長さを指定するには、ACBP フィールドの B 属性ビット (SEG ATTR の項を参照してください) を使わなければなりません。

SEGMENT NAME INDEX (セグメント名インディックス)

セグメント名はプログラマー、またはトランスレータがセグメントにつける名前です。たとえば CODE, DATA, TAXDATA, MODULENAME, CODE, STACK 等です。このフィールドは LNAME レコードが与える名称リストにインデックス付けすることでセグメント名になります。

CLASS INDEX (クラス名インディックス)

クラス名はプログラマやトランスレータがセグメントに割り当てる名前です。割り当てられていない場合に、名前は空になり、長さは0になります。クラス名の目的は、MAS 中の LSEG の順序づけに使うハンドルを(プログラマが)定義できるようにするためです。たとえば RED, WHITE, BLUE; ROM, FASTRAM, DISPLAYRAM 等です。このフィールドは、LNAME レコードの与える名称リストにインデックス付けすることによってクラス名を与えます。

OVERLAY NAME INDEX (オーバーレイ名インディックス)**注意：**

この項目は、V2.40 以降のマイクロソフト LINK では無視されますが、それ以前のバージョンには採用されています。ただしインテル仕様とは意味が異なります。

オーバーレイ名は、プログラマーの要求により、オーバーレイ名はトランスレータ、またはマイクロソフト LINK がセグメントにつける名前です。クラス名と同様、オーバーレイ名は空であってもかまいません。このフィールドは LNAME レコードが与える名称リストにインデックス付けすることによりオーバーレイ名を与えます。

注意：

セグメントの「完全な名称」とはセグメント名、クラス名、オーバーレイ名3つの部分から成る名前です。(後半の2つの名前は空とすることができます)。

6.12.5 GROUP DEFINITION RECORD (グループ定義レコード)
(GRPDEF)

REC TYP 9AH (1)	RECORD LENGTH (2)	GROUP NAME INDEX (1以上)	GROUP COMPONENT DESCRIPTOR (1以上)	CHK SUM (1)
			REP	

GROUP NAME INDEX (グループ名インデックス)

グループ名は LSEG が参照される時に使う名前です。このグループの重要な特質として、結果的に LSEG が MAS 中で固定される時、グループの各 LSEG をカバーするフレームが存在しなくてはならないことが挙げられます。

GROUP NAME INDEX フィールドは LNAME レコードが与える名前のリストにインデックス付けすることによってグループ名を与えます。

GROUP COMPONENT DESCRIPTOR (グループ要素記述子)

各 GROUP COMPONENT DESCRIPTOR のフォーマットをつぎに示します。

SI (FFH) (1)	SEGMENT INDEX (1 以上)
--------------------	----------------------------

記述子の第1バイトは 0FFH であり、前にある SEGDEF レコードが記述する LSEG を選択する SEGMENT INDEX フィールド1つを含みます。

インテルは他にも4つのグループ記述タイプと各々の意味を定義しています。これらは 0FFH, 0FDH, 0FBH, 0FAH です。マイクロソフト LINK は、これらすべてを 0FFH と同一として扱います(つまり、常に 0FFH にはセグメントインデックスが続くものとし、実際に値が 0FFH であるか否かをチェックしません)。

6.12.6 TYPE DEFINITION RECORD (型定義レコード)
(TYPDEF)

REC TYP 8EH (1)	RECORD LENGTH (2)	NAME (常にヌル) (1 以上)	EIGHT LEAF DESCRIPTOR (1 以上)	CHK SUM (1)
REP				

マイクロソフト LINK は TYPDEF レコードを共有変数の位置づけにのみ使用します。これはインテルが目的としたものではありません。6.14 の「共有変数の型に関するマイクロソフト表現法」を参照してください。

必要な数の EIGHT LEAF DESCRIPTOR (8 リーフ記述子) フィールドを使って分岐を記述します (最後のレコードを除く)。この最後のレコードは 1~8 リーフを記述します。

可変の型インデックスの値 (1~32767) は他のレコードタイプに (オブジェクトタイプとオブジェクト名を関連づけるために) 含まれていますが, オブジェクトファイル中で TYPDEF レコードを記述する順序によって暗黙の内に定義されます。

NAME (名前)

このフィールドの使用は予約されています。トランスレータはこのフィールドを 0 に (長さが 0 の名前の表現) しておきます。

EIGHT LEAF DESCRIPTOR (8 リーフ記述子)

このフィールドは 8 つまでのリーフを記述することができます。

E N (1)	LEAF DESCRIPTOR (1 以上)
	RPT

EN フィールドは 1 バイトつまり 8 ビットで、(左から右の順に) 8 つのリーフが容易 (ビット=0)、または精密 (ビット=1) であることを示すものです。

1~8 個の LEAF DESCRIPTOR (リーフ記述子) のフォーマットはつぎのうちのいずれかになります。

0~128 (1)

129 (1)	0~64K-1 (2)
------------	----------------

132 (1)	0~16M-1 (2)
------------	----------------

136 (1)	$-2G+1$ $\}$ $2G-1$ (2)
------------	----------------------------------

第1のフォーマット(1バイト)は0~127の値を持ち、与えられた数値を値とする数字リーフを表現します。

第2のフォーマットは、先行バイトとして129を、数字リーフを表現します。数値は続く2バイトに含まれます。

第3のフォーマットは、先行バイトとして132を、数字リーフを表現します。数値は3バイトに含まれます。

第4のフォーマットは、先行バイトとして136があり、符号付き数字リーフを表現します。数値は続く4バイトに含まれ、必要に応じて符号が付けられます。

6.12.7 PUBLIC NAMES DEFINITION RECORD

(パブリック名定義レコード)

(PUBDEF)

REC TYP 90H (1)	RECORD LENGTH (2)	PUBLIC BASE (1以上)	PUBLIC NAME (1以上)	PUBLIC OFFSET (2)	TYPE INDEX (1以上)	CHK SUM (1)
RPT						

このレコードは単一、または複数の PUBLIC NAME のリストを与えますが、各名前ごとに3つのデータがあります。(1) 名前のベース、(2) 名前のオフセット値、(3) 名前の表現する実質の型

PUBLIC BASE (名前のベース値)

PUBLIC BASE のフォーマットはつぎのとおりです。

GROUP INDEX (1以上)	SEGMENT INDEX (1以上)	FRAME NUMBER (2)
COND		

GROUP INDEX フィールドのフォーマットはすでに述べてあり、0～32767 の値を取ります。0 でないグループインデックスはパブリックシンボルのついたグループに結び付き、6.8 の“フィックスアップのためのフレームの概念”の F2c の方法で使用されます。グループインデックスが0の場合、関連グループがないことを示しています。

SEGMENT INDEX フィールドのフォーマットもすでに説明したように 0～32767 の値を取ります。

0 でないセグメントインデックスは1つの LSEG を指定します。この場合、レコード中で定義される各パブリックシンボルのロケーションは、選択した LSEG の第1バイトからの負でない変位 (PUBLIC OFFSET フィールドで指定します) として扱われ、フレーム番号はつけられません。

セグメントインデックス（グループインデックスが0のときのみ有効）が0の場合、レコード中で定義されているパブリックシンボルのロケーションは、フレーム番号フィールドの値が定義するフレームのベースからの変位とされます。

セグメントインデックスおよびグループインデックスの双方が0である時のみ、フレーム番号が存在します。

0以外のグループインデックスはあるグループを指定します。このグループは、このレコード中で定義されるすべてのパブリックシンボルを参照のための「参照のフレーム」とされます。つまりマイクロソフト LINK はつぎの動作を行います。

1. つぎに示す形式のフィックスアップ

TARGET: EI (P)

FRAME: TARGET

（この場合の「P」はこの PUBDEF レコード中のパブリックシンボルです）はマイクロソフト LINK によってつぎの形式のフィックスアップに変換されます。

TARGET: SI (L), d

FRAME: GI (G)

この場合の「SI (L)」と「d」はセグメントインデックスと PUBLIC OFFSET フィールドによって与えられます。（正常な動作では新しいフィックスアップ中のフレーム指定子を古いフィックスアップ（FRAME: TARGET）と同一視します。

2. セグメントインデックス、パブリックオフセットとしてパブリックシンボルの値が定義され、（オプションで）フレーム番号フィールドが {ベース, オフセット} の対に変換される時、ベース部分は示されたグループのベースとされます。ここで0以外の16ビットオフセットが、パブリックシンボル値の定義を満足しない場合はエラーになります。

グループインデックスが0の場合、グループを指定しません。マイクロソフト LINK はシンボルを参照するフィックスアップのフレーム指定を変更することはありません。そしてマイクロソフト LINK はこれをパブリックシンボルの絶対値のベース部分を、セグメントインデックスフィールドによって決定されるセグメント（LSEG または PSEG）の正規フレームとします。

PUBLIC NAME (パブリック名)

PUBLIC NAME フィールドは、オブジェクトの名前を与えます。そして、そのオブジェクトの MAS 中のロケーションは他のモジュールに使用可能になります。名前は1つ以上の文字を含まなければなりません。

PUBLIC OFFSET (パブリックオフセット)

PUBLIC OFFSET フィールドは 16 ビット値で、LSEG(セグメントインデックス>0 の場合) に対応したパブリックシンボルのオフセット、または指定したフレーム (セグメントインデックス=0 の場合) に対応したパブリックシンボルのオフセットです。

TYPE INDEX (型 INDEX)

TYPE INDEX フィールドはパブリックシンボルの表す実質の型の記述子を含む単一の前にある TYPDEF (型定義) レコードを識別します。リンカはこのフィールドを無視します。

6.12.8 EXTERNAL NAMES DEFINITION RECORD

(エクスターナル名定義レコード)

(EXTDEF)

REC TYP 8CH (1)	RECORD LENGTH (2)	EXTERNAL NAME (1 以上)	TYPE INDEX (1 以上)	CHK SUM (1)
RPT				

このレコードはエクスターナル名のリストおよび、各名前について、名前の表現するオブジェクトの型を与えます。マイクロソフト LINK は各エクスターナル名に相当するパブリック名 (存在する時は) の与える値を割り当てます。

EXTERNAL NAME (エクスターナル名)

このフィールドはエクスターナルオブジェクトの名前 (長さが 0 であってはならない) を与えます。

エクスターナル名レコードに名前を含めることは、パブリックシンボルとして宣告された同一の名称を含むモジュールにオブジェクトファイルをリンクするための暗黙の要求です。この要求はエクスターナル名が何かの FIXUPP レコードによって参照されるか否かによって発生します。

モジュール中で EXTDEF レコードの順序づけは、各 EXTDEF レコード中のエクスターナル名の順序づけと共にモジュールによって要求される、すべてのエクスターナル名配置の順序が発生します。よってエクスターナル名は 1, 2, 3, 4……と番号づけされます。この番号は FIXUPP レコードの TARGET DATUM や、または FRAME DATUM フィールドの「エク

スターナルインデックス」¹として、特定のエクスターナル名を参照するために使われます。

注意：

8086 のエクスターナル名は 1, 2, 3……と確実に番号がつけられています。この点は 8080 のエクスターナル (外部) 名の番号が 0, 1, 2, ……と 0 から始まっていた点と異なります。これは特定の意味を持つデフォルト値として 0 を使う、他の 8086 インデックス (セグメントインデックス、型インデックス等) を考慮したためです。

エクスターナルインデックスは前方に向かって参照することはありません。たとえば K 番目のオブジェクトを定義するエクスターナル定義レコードは、そのオブジェクトをインデックス K で参照するすべてのレコードの前に置かれます。

TYPE INDEX (型インデックス)

このフィールドは前にある、エクスターナルシンボルによって名前付けされたオブジェクトの型の記述子を含む1つの TYPDEF (型定義) レコードを識別するものです。

型インデックスはマイクロソフト LINK では、共有変数の割り振りにのみ使われます。

6.12.9 LINE NUMBER RECORD (行番号レコード)
(LINNUM)

REC TYP 94H (1)	RECORD LENGTH (2)	LINE NUMBER BASE (1以上)	LINE NUMBER (2)	LINE NUMBER OFFSET (2)	CHK SUM (1)
			RPT		

このレコードはソースコード中の行番号と、それに対する翻訳されたコードの対応づけの手段をトランスレータに与えるものです。

LINE NUMBER BASE (行番号ベース)

行番号ベースは、つぎの形式を取ります。

GROUP INDEX (無視される) (1 以上)	SEGMENT INDEX (1 以上)
-------------------------------------	--------------------------------

セグメントインデックスは、あるソースの行番号に対応する先頭のバイトのロケーションを決定します。

LINE NUMBER (行番号)

0～32767 の行番号を 2 進法で与えます。高位ビットは他の目的で使用するために予約されており、0 になっています。

LINE NUMBER OFFSET (行番号オフセット)

LINE NUMBER OFFSET フィールドは 16 ビット値で行番号 LSEG に対応したオフセットです。(セグメントインデックス > 0 の時)

6.12.10 LOGICAL ENUMERATED DATA RECORD

(論理エヌメリテッドデータレコード)

(LEDATA)

REC TYP A0H (1)	RECORD LENGTH (2)	SEGMENT INDEX (1 以上)	ENUMERATED DATA OFFSET (2)	DAT (1)	CHK SUM (1)
RPT					

このレコードは、8086 メモリイメージの一部を構成する連続データを与えます。

SEGMENT INDEX (セグメントインデックス)

このフィールドは0であってはならず、(LEDATA RECORDの前に置かれた)セグメント定義レコードに関するインデックスを指定します。

ENUMERATED DATA OFFSET

このフィールドは、(セグメントインデックスで指定される)LSEGのベースに関してのオフセットを指定し、DATフィールドの第1バイトの相対ロケーションを定義します。DATフィールドの連続したデータバイトはメモリの高位ロケーションを連続して占めます。

DAT

このフィールドはリロケータブル、または絶対データの連続した(最大1024までの)バイトを与えます。

6.12.11 LOGICAL ITERATED DATA RECORD

(論理アイテラテッドデータレコード)

(LIDATA)

REC TYP	RECORD LENGTH	SEGMENT INDEX	ITERATED DATA OFFSET	ITERATED DATA BLOCK	CHK SUM
A2H					
(1)	(2)	(1以上)	(2)	(1以上)	(1)

└────────── RPT ─────────┘

このレコードは、8086 メモリイメージの一部を構成する連続データを与えます。

SEGMENT INDEX (セグメントインデックス)

このフィールドは0であってはならず、(LIDATA レコードの前に置かれた)SEGDEFレコードに関するインデックスを指定します。

ITERATED DATA OFFSET

このフィールドは、(セグメントインデックスで指定される)LSEGのベースに関してのオフセットを指定し、ITERATED DATA BLOCKの第1バイトの相対ロケーションを定義します。ITERATED DATA BLOCKの連続したデータバイトは、メモリの高位ロケーションを連続して占めます。

ITERATED DATA BLOCK

この反復したフィールドは反復するデータバイトを指定するための構造です。
この構造のフォーマットをつぎに示します。

REPEAT COUNT	BLOCK COUNT	CONTENT
(2)	(2)	(1以上)

注意：

リンクは、ITERATED DATA BLOCKの大きさが512バイトを超える LIDATA レコードを扱うことはできません。

REPEAT COUNT

このフィールドは ITERATED DATA BLOCK の CONTENT の部分の反復回数を指定します。REPEAT COUNT は 0 であってはなりません。

BLOCK COUNT

このフィールドは、この ITERATED DATA BLOCK の CONTENT 部にある ITERATED DATA の BLOCK COUNT を指定します。このフィールドの値が 0 である場合、ITERATED DATA BLOCK の CONTENT 部はデータバイトとして解釈されます。0 以外の場合、CONTENT 部には ITERATED DATA BLOCK がその数だけ繰り返されます。

CONTENT

このフィールドは、前の BLOCK COUNT フィールドの値にしたがって、2 つの方法のうちの一方で解釈されます。

BLOCK COUNT が 0 である場合、このフィールドは 1 バイトのカウントと、そのカウントによって数が示されるデータバイトになります。

BLOCK COUNT が 0 以外の場合、このフィールドは別の ITERATED DATA BLOCK の第 1 バイトとして解釈されます。

注意：

一番外のレベルから数えて、ネスト(入れ子)されている ITERATED DATA BLOCK の数は 17 以下に制限されています。つまり反復レベル数は 17 以下に限定されています。

6.12.12 FIXUP RECORD (フィックスアップレコード) (FIXUPP)

REC TYP 9CH (1)	RECORD LENGTH (2)	THREAD or FIXUP (1以上)	CHK SUM (1)
RPT			

このレコードは0かそれ以上のフィックスアップを指定します。各フィックスアップは、前にある DATA レコード中のロケーションに対して変更（フィックスアップ）を要求します。DATA レコードは、それを参照する1つ以上のフィックスアップレコードに従えることができます。各フィックスアップは、ロケーション、モード、ターゲット、フレームの4つのデータを指定する FIXUP フィールドによって指定されます。フレームとターゲットは完全にフィックスアップファイルが指定されるか、または前の THREAD フィールドを参照することで指定されます。

THREAD フィールドは、ターゲットまたはフレームを識別するために、その後に参照されるデフォルトターゲット、またはデフォルトフレームを指定します。フレーム指定のために4つ、ターゲット指定のために4つの計8つの THREAD(スレッド)が指定されます。スレッドによって一度、ターゲットおよびフレームが指定されると、型（ターゲットまたはフレーム）とスレッド番号(0~3)が同一の THREAD フィールドが(同じレコード、または他の FIXUPP レコード中で) 現れるまで後続く FIXUP フィールドによって参照されます。

THREAD (スレッド)

THREAD フィールドのフォーマットはつぎのとおりです。

TRD (1)	INDEX (1以上)
COND	

TRD DAT (ThReaD DATa: スレッドデータ) サブフィールドはつぎの内部構造をもつバイトです。

0	D	Z	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> METHOD </div> <div style="text-align: center;"> THRED </div> </div>
---	---	---	--

「Z」は1ビットのサブフィールドで、現在、機能が定義されておらず、0である必要があります。

「D」サブフィールドは指定されているスレッド型を識別する1ビットです。D=0の場合はターゲットスレッドが定義されていますが、D=1の場合はフレームスレッドが定義されています。

METHODは0~3 (D=0の場合)、または0~6 (D=1の場合) を取る3ビットのサブフィールドです。

D=0の場合、METHODは、(0, 1, 2, 3, 4, 5, 6, 7)を4で割った余りの値を取ります。ここに0, …… , 7が6.8に示すターゲットを指定する方法T0, …… , T7を示しています。このようにMETHODは、第1または第2の方法でターゲットが指定されたか否かを示すことなく、ターゲット指定に必要なインデックスやフレーム番号の種類を示します。方法2b, 3, 7はマイクロソフトLINKでは使えないことに注意してください。

D=1の場合、METHOD=0, 1, 2, 4, 5は、フレームを指定する方法F0, …… , に対応します。ここで、METHODはフレームを指定するのに必要なインデックス (存在する場合は) の種類を示します。方法3と、5dはマイクロソフトLINKでは使えないことに注意してください。

スレッドは0~3の数で、スレッドフィールドによって定義されるフレームまたはターゲットのスレッド番号と結びつきます。

インデックスは、METHODサブフィールド中の指定によってセグメントインデックス、グループインデックス、またはエクスターナルインデックスになります。このサブフィールドはMETHODにF4, または F5が指定されている場合は存在しません。

FIXUP (フィックスアップ)

FIXUP フィールドのフォーマットをつぎに示します。

LOCAT (2)	FIX DAT (1)	FRAME DATUM (1 以上)	TARGET DATUM (1 以上)	TARGET DISPLACEMENT (1 以上)
		COND	COND	COND

LOCAT はつぎに示すフォーマットを持つ2バイトです。

1	M	S	LOC	DATA RECORD OFFSET
			LOBYTE	HIBYTE

「M」はフィックスアップのモード（自己相対（M=0）、セグメント相対（M=1））を指定する1ビットサブフィールドです。

注意：

LIDATA レコードには自己相対フィックスアップが適応できない場合もあります。

「S」はターゲット DISPLACEMENT サブフィールドの長さを指定する1ビットサブフィールドです。FIXUP フィールド中に TARGET DISPLACEMENT が存在する（以下参照）場合は、2バイト（16ビットの負でない数、S=0）、または3バイト（24バイト数の2の補数、S=1）の値を取ります。

注意：

3バイトサブフィールドは将来の拡張により存在し得ますが、現在は使用されていません。したがって、現在はS=0に強制されます。

LOC は、フィックスアップされる先行 DATA レコード中のバイトが何であることを示す3ビットのサブフィールドで、LOC=0の場合、「低位バイト」、LOC=1の場合「オフセット」、LOC=2の場合「ベース」、LOC=3の場合「ポインタ」、LOC=4の場合「高位バイト」に、それぞれなります。LOCの他の値は無効です。

DATA RECORD OFFSET は0～1023をとる数で、先行する DATA レコード中の低位

バイトのロケーション(フィックスアップされる実際のバイト)の相対位置を与えます。DATA RECORD OFFSET は DATA レコード中のデータフィールドの第1バイトと対応します。

注意：

先行する DATA レコードが LIDATA レコードである場合、DATA RECORD OFFSET の値が、ITERATED DATA フィールドの REPEAT COUNT サブフィールド、または BLOCK COUNT サブフィールド中の「ロケーション」を示すこともあり得ます。しかし、このような参照はエラーになります。このような無効レコードに対してマイクロソフト LINK の動作は不定となります。

FIX DAT バイトのフォーマットをつぎに示します。

F	 FRAME 	T	P	 TARGET
	(注1)			(注2)

注1：フレーム指定方法 2b, F3, F5d は使えません。

注2：ターゲット方指定方法 T3, T7 は使えません。

「F」は、このフィックスアップのフレームがスレッドによって指定される (F=1) か、または明確に指定するか (F=0) を与える 1 ビットサブフィールドです。

フレームは F ビットで示されるどちらかの方法によって解釈される数です。F が 0 の場合、フレームはフレーム指定方法 F0, ……、F5 に対応する 0～5 の数です。F=1 の場合、フレームはスレッド番号 (0～3) です。これは、同一スレッド番号のついたフレームスレッドを定義する THREAD フィールドによって、最も最近に定義されたフレームを指定します (THREAD フィールドは同一の、または先行する FIXUPP レコード中に存在します。)

「T」はこのフィックスアップに指定されるターゲットがスレッド参照によって定義される (T=1) か、または FIXUP フィールド中で明確に指定される (T=0) かを示す 1 ビットのサブフィールドです。

「P」はターゲットが第1の方法で指定される (TARGET DISPLACEMENT が必要、P=0) か、または第2の方法で指定される (TARGET DISPLACEMENT が不要、P=1) かを示す 1 ビットのサブフィールドです。ターゲットスレッドは第1／第2属性を持たぬため、P ビットはターゲット指定の第1／第2属性を与える唯一のフィールドです。

ターゲットは 2 ビットのサブフィールドとして解釈されます。T=0 の場合、ターゲットフィールドは、P の値によって (P は T0, ……、T7 の高位ビットとして解釈されます) 方法 T0,

……, T3, または T4, …… , T7 に対応する 0~3 の数を与えます。スレッドによってターゲットを指定する場合 (T=1), ターゲットはスレッド番号 (0~3) を指定します。

FRAME DATUM はフレーム指定の「参照」部で、セグメントインデックス、グループインデックス、エクスターナルインデックスのいずれかです。FRAME DATUM サブフィールドは、フレームがスレッドによって指定されず(F=0), 方法 F4, F5, F6 によって明示されない場合にのみ存在します。TARGET DATUM は、ターゲット指定の「参照」部で、セグメントインデックス、グループインデックス、エクスターナルインデックスまたはフレーム番号のいずれかです。

TARGET DATUM サブフィールドは、ターゲットがスレッドによって指定されない時 (P=0) のみ存在します。

TARGET DISPLACEMENT はターゲットを指定する「第1の」方法が要求する 2 バイトの変位です。この 2 バイトサブフィールドは P=0 の時のみ存在します。

注意：

この方法については、すべて 6.8 の “フィックスアップのためのフレームの概念” に説明してあます。

6.12.13 MODULE END RECORD (モジュールエンドレコード)
(MODEND)

REC TYP	RECORD LENGTH	MOD TYP	START ADDRS	CHK SUM
8AH				
(1)	(2)	(1)	(1 以上)	(1)

COND

このレコードのオブジェクトは 2 つあります。このレコードはモジュールの終了を示し、終了したばかりのモジュールに実行開始のエントリポイントが指定されているか否かを示します。後者が存在する場合、実行アドレスも指定します。

MOD TYP

このフィールドはこのモジュールの属性を示します。ビット割り当てに付帯する意味はつぎのとおりです。

 MATTER 	Z	Z	Z	Z	Z	L
----------------	---	---	---	---	---	---

MATTER はつぎに示す、モジュール属性を指定する 2 ビットサブフィールドです。

<u>MATTER</u>	<u>モジュール特性</u>
0	非メインモジュールにスタートアドレスなし
1	非メインモジュールにスタートアドレスあり
2	メインモジュールにスタートアドレスなし
3	メインモジュールにスタートアドレスあり

「L」は START ADDR フィールドがマイクロソフト LINK によるフィックスアップが必要な論理アドレスとして解釈される (L=1) か否かを示します。また、マイクロソフト LINK では L は常に 1 に固定されることに注意してください。

「Z」はそのビットに現在機能割り付けられていないことを示します。このビットは 0 である必要があります。

物理開始アドレス (L=0) は使用できません。

START ADDR フィールド (MATTER が 1, および 3 の時のみ存在) のフォーマットはつぎのとおりです。

START ADDRESS

END DAT (1)	FRAME DATUM (1以上)	TARGET DATUM (1以上)	TARGET DISPLACEMENTSUM (2)
	COND	COND	COND

モジュールの開始アドレスは、モジュール中に存在する他の論理参照のすべての属性を持ちます。

論理開始アドレスから物理開始アドレスへのマッピングは、他の(フィックスアップや FIXUPP レコードの解説で述べたような)論理開始アドレスから物理アドレスへのマッピングと全く同様の方法で行われます。START ADDRS フィールドの前述のサブフィールドは FIXUPP レコード中の FIX DAT, FRAME DATUM, TARGET DATUM, TARGET DISPLACEMENT フィールドと同じ意味を持っています。「第1」フィックスアップのみが許されています。フレーム指定方法 F4 は認められません。

6.12.14 COMMENT RECORD (コメントレコード) (COMENT)

REC TYP 88H (1)	RECORD LENGTH (2)	COMMENT TYPE (2)	COMMENT (1以上)	CHK SUM (1)
--------------------------	-------------------------	------------------------	------------------	-------------------

このレコードによってトランスレータは、オブジェクトにコメントを含めることができます。

COMMENT TYPE

このフィールドはこのレコードの持つコメントの型を示します。これによりコメントに対して、選択的に動作するような手段に対して、コメントを構成することができます。このフィールドのフォーマットはつぎのとおりです。

N	N							COMMENT
P	L	Z	Z	Z	Z	Z	Z	CLASS

NP(NOPURGE：除去なし)ビットが1である場合は、COMENTレコードを削除する能力を持つオブジェクトファイルユーティリティプログラムによっても、除去することができないことを示します。

NL(NOLIST：リストなし)ビットが1である場合は、オブジェクトCOMENTレコードのリスト能力を持つオブジェクトファイルユーティリティプログラムのリスティングファイル中に、COMMENTフィールドの文章をリストすることができないことを示します。

COMMENT CLASS フィールドの定義をつぎに示します。

0	言語トランスレータコメント。
1	インテル著作権コメント。 NP ビット設定しなければなりません。
2~155	インテル使用のために予約。 つぎの注意：1を参照してください。
156~255	ユーザーのために予約。 インテル社の製品に対して、これらの中の値は意味を持ちません。 つぎの注意：2を参照してください。

COMMENT

このフィールドはコメント情報を与えます。

注意：1

クラス値 129 はリンカのライブラリ検索リストに加えるためのライブラリの指定に使います。この場合、COMMENT フィールドはライブラリ名を含みます。すべての他の名称指定と異なり、ライブラリ名にはその長さがつけられていないことに注意してください。その長さはレコード長によって決定されます。「NODEFAULTLIBRARYSEARCH」スイッチによって、リンカはクラス値が 129 の COMMENT レコードをすべて無視します。

注意：2

クラス値 156 は MS-DOS レベル番号の指定に使用します。クラス値が 156 の時、COMMENT フィールドには MS-DOS レベル番号を指定する 2 バイト整数が含まれます。

6.13 レコードの番号によるリスト

- * 6E RHEADR
- * 70 REGINT
- * 72 REDATA
- * 74 RIDATA
- * 76 OVLDEF
- * 78 ENDREC
- * 7A BLKDEF
- * 7C BLKEND
- * 7E DEBSYM
- 80 THEADR
- * 82 LHEADR
- * 84 PEDATA
- * 86 PIDATA
- 88 COMENT
- 8A MODEND
- 8C EXTDEF
- 8E TYPDEF
- 90 PUBDEF
- * 92 LOCSYM
- 94 LINNUM
- 96 L NAMES
- 98 SEGDEF
- 9A GRPDEF
- 9C FIXUPP
- * 9E (none)
- A0 LEDATA
- A2 LIDATA
- * A4 LIBHED

- * A6 LIBNAM
- * A8 LIBLOC
- * AA LIBDIC

注意：

(*) が文頭についたレコード型はマイクロソフト LINK では使えません。オブジェクトモジュール中にあった場合も無視されます。

6.14 共有変数の型に関するマイクロソフト表現法

本章は 8086 と 80286 (80286 と上位互換性のあるものを含む) 上での共有変数割り振りに関するマイクロソフト規格を定義します。

共有変数は、最終サイズと最終ロケーションがコンパイル時に固定されない、初期化されないパブリック変数です。相互にリンクされる複数のモジュール中に、共有変数が宣言されていて、いくつかの宣言中で指定された最大サイズとその実効サイズが等しい時、共有変数は FORTRAN の共有ブロックのようなものになります。また、C 言語では、初期化されていないパブリック変数は共有変数です。つぎに、C 言語による同一の共有変数の異なる宣言の例を示します。

```
char foo [4];      /* In file a, c */
char foo [1];      /* In file b, c */
char foo [1024];   /* In file c, c */
```

a. c, b. c, c. c によって作成されたオブジェクトが相互にリンクされている場合、リンクは文字アライメント「foo」に 1024 バイトを割り当てます。

オブジェクトテキスト中で、エクスターナル定義レコード (EXTDEF) とそれが参照する型定義レコード (TYPDEF) によって共有変数が定義されます。

共有変数に対する TYPDEF のフォーマットはつぎのとおりです。

REC TYP 8EH (1)	RECORD LENGTH (2)	0 (1)	EIGHT LEAF DESCRIPTOR (1 以上)	CHK SUM (1)
--------------------------	-----------------------------	--------------	---------------------------------------	-----------------------

EIGHT LEAF DESCRIPTOR (8 リーフ記述子) フィールドのフォーマットはつぎのとおりです。

E N (1 以上)	LEAF DESCRIPTOR (1 以上)
------------------	------------------------------

EN フィールドは LEAF DESCRIPTOR フィールド中の次の 8 つのリーフが EASY (単純) であるか (ビット=0), NICE (精密) であるか (ビット=1) を指定します。共有変数の TYPDEF では、このバイトは常に 0 です。

LEAF DESCRIPTOR フィールドはつぎの 2 つのフォーマットのうちいずれかを取ります。

デフォルトのデータセグメント中の (near 変数) 共有変数フォーマットはつぎのとおりです。

NEAR 62H (1)	VAR TYP (1)	LENGTH IN BITS (1 以上)	VAR SUBTYP (1 以上)
			(OPTIONAL)

VARTYP (変数型) フィールドは SCALAR (スカラ; 7BH), STRUCT (構造体; 79H)

または ARRAY (配列; 77H) のいずれかです。VAR SUBTYP フィールドはリンクに無視されます。

デフォルトのデータセグメント中になく共有変数のフォーマットはつぎのとおりです。

FAR 61H (1)	VAR TYP 77H (1)	NUMBER OF ELEMENTS (1 以上)	ELEMENT TYP INDEX (1 以上)
-------------------	--------------------------	------------------------------------	-----------------------------------

この VARTYP (変数型) フィールドは ARRAY (77H) に限られます。RECORD LENGTH フィールドによって NUMBER OF ELEMENTS を指定し、ELEMENT 型インデックスのフォーマットがその (near) 共有変数の形をしている定義済みの TYPDEF のインデックスになります。

LENGTH IN BITS や NUMBER OF ELEMENTS フィールドのフォーマットは、本マニュアルの TYPDEF レコードフォーマットの所で説明した LEAF DESCRIPTOR フィールドのフォーマットと同一です。

リンク時間の意味

先行して記述されたフォーマットのうちの 1 つの TYPDEF を参照する、すべての EXT DEF は共有変数として扱われます。他はすべて整合パブリックシンボル定義 (PUBDEF) を持つはずのエクスターナル定義シンボルとして扱われます。共有変数定義に整合する PUBDEF は共有変数をオーバーライドします。2 つの共有変数定義は、定義の中で与えられる名前が整合する時、一致するといえます。2 つの整合する定義が、共有変数が near, far にかかわらず一致しない時は、リンクは変数が near であると仮定します。

変数が near である場合、指定されたサイズのうちで、そのサイズを最大とします。変数が far である場合、リンクはアライメント (配列) 要素のサイズ指定に矛盾がある時は警告を表示します。このような矛盾がない場合は変数のサイズは要素サイズに指定された最大要素数をかけたものになります。すべての near 変数のサイズの合計は 64K バイトを超えられません。すべての far 変数のサイズの合計は、その機械のアドレス指定可能メモリ空間を超えることはできません。

「HUGE」共有変数

64K バイトを超えるサイズを持つ far 変数は連続したセグメント中 (8086) かまたは連続選択

装置 (80286) 中に置かれます。セグメント中には、huge 共有変数は他のデータ項目を置きません。

リンカが大きな共有変数と near 共有変数を整合させる定義を見つけると、警告メッセージを発します。near 変数は 64K バイトより大きいことはあり得ないからです。

第7章

プログラムヒント

7.1 イントロダクション

本章では、将来の MS-DOS のバージョンに対応するための V 3.3 のプログラム手順について解説します。

7.2 割り込みタイプ

割り込みタイプ 22 H (終了アドレス) は、絶対にユーザーが実行してはいけません。この割り込みタイプは、MS-DOS 自身だけが実行できます。

割り込みタイプ 24 H (致命的エラーによる中断アドレス) は注意して使用してください。

割り込みタイプ 24 H ハンドラはシステムコールの 01 H ~ 0CH, 30 H, 59 H についてのみ実行できます。これ以外のコールを行うとスタックが破壊され、「再試行する」または「無視する」を選択した場合の処理が正しく行われなくなります。

割り込みタイプ 24 H ハンドラは ES レジスタを保存しなければなりません。また、プログラムを「再試行する」か、「無視する」を選択した時、レジスタ SS, SP, DS, BX, CX, DX を保存します。

割り込みタイプ 24 H は選択の回答を受け取ると、回答を伴って、IRET によって MS-DOS に戻ります。

割り込みタイプ 24 H で IRET を実行しないプログラムでは、01 H から 0CH 以外のコールをするまでシステムは不安定な状態となります。「無視する」を選択した場合、不正なデータや無効なデータが内部システムバッファに残ります。

割り込みタイプ 23 H (CTRL-C の抜け出しアドレス) と割り込みタイプ 24 H (致命的エラーによる中断アドレス) のトラップは避けてください。割り込みタイプ 24 H によるトラップエラーをコピー保護などの目的で使ってはいけません。この方法は、将来の MS-DOS のバージョンで使用できなくなる可能性があります。

割り込みタイプ 23H (CTRL-C の抜け出しアドレス) は、絶対にユーザーが実行してはいけません。この割り込みタイプは、MS-DOS 自身だけが実行できます。

プログラムが割り込みタイプ 25 H (アブソリュートディスクリード) または 26 H (アブソリュートディスクライト) を実行する前に、すべてのレジスタをセーブしてください。

これらの割り込みは、セグメントレジスタを除くすべてのレジスタを破壊します。

メモリに、またはメモリから割り込みベクタを直接、書き込みまたは読み出しすることは避けてください。

ファンクション 25 H (割り込みベクタをセットする) と 35 H (割り込みベクタを得る) で割り込みテーブル中の値を得る、またはセットすることができます。

7.3 システムコール (ファンクションリクエスト)

新しいシステムコールを使います。プログラムが MS-DOS の V 2.0 以前と互換性を保つ必要のある場合を除いて、システムコールは新しい方を使ってください。詳細については 1.8 の “V 2.0 以前のシステムコール” を参照してください。

ファンクション 01 H から 0 CH と 26 H (新しい PSP を作成する) を使うことは避けてください。標準入出力の読み出し、書き込みには新しいシステムコールを使用してください。子プロセスを起動するときはファンクション 26 H の代わりにファンクション 4 BH、コード 00 H (プログラムのロードと実行) を使います。

複数の処理を行っているときは、ファイルシェアリングのシステムコールを使います。詳細については、1.5.2 の “ファイル管理のファンクションリクエスト” を参照してください。

MS-Networks には、ネットワークのシステムコールを使います。IOCTL の様式のいくつかは、MS-Networks 用に用意されたものです。詳細については、1.6 の “MS-Networks” を参照してください。

ファンクション 0EH (ディスクの選択) によってディスクを選択する時は、AL に返された値を注意して扱います。AL 中の値は論理ドライブの最大数を返しますが、どのドライブが有効であるかは示しません。

7.4 デバイス管理

インストール可能なデバイスドライバ (装置ドライバ) を使います。MS-DOS は、BIOS を追加できる構造を持つため、CONFIG.SYS に登録することによって、ブート時にデバイスドライバをインストールすることができます。転送するデータの単位は、ブロックデバイスドライバは一度に1ブロック、キャラクタデバイスドライバは1バイトです。

この2つのデバイスドライバの例は2章の“MS-DOS デバイスドライバ”を参照してください。

デバイスドライバはバッファリングされた I/O を使います。データストリームは 64K バイトまでバッファリングすることができます。

大量の出力を画面に送る場合に、1回のコールでこれを行うことができ、効率が上がります。

ファンクション 06H と 07H (直接コンソール I/O と直接コンソール入力) を使用し、直接コンソールと I/O を行うプログラム、<CTRL-C> をデータとして読み取るプログラムは、“CTRL-C の検査” がオフになっていることを確認する必要があります。

プログラムはこの“CTRL-C の検査” がオフになっていることを、ファンクション 33H を使って確認します。

7.5 メモリ管理

■ メモリ管理を使います。

MS-DOS は各メモリ領域の先頭にメモリコントロールブロックを置くことで割り付けられたメモリを管理します。プログラムはファンクション 48H (メモリの割り当て)、49H (割り当てられたメモリの解放)、4AH (割り当てられたメモリブロックの変更) を使って不必要なメモリを解放します。

このことは将来のバージョンに対し、互換性を保つのに有効です。

メモリ管理の詳細については、1.3 の“メモリ管理”を参照してください。

■ 割り付けされたメモリだけを使います。

システムコールのメモリ管理によって得られたものでないメモリを直接アクセスしてはいけません。絶対アドレス指定ではなく、相対アドレス指定のみを使用します。

プログラムが割り当てられていないメモリを使用した場合、他のメモリ管理を破壊したり、他のアプリケーションを失敗に終わせます。

7.6 プロセス管理

EXEC ファンクションコールでプログラムのロードと実行を行います。プログラムのロードおよびオーバーレイには EXEC ファンクション (ファンクション 4BH) を使います。EXE 形式のファイルのヘッダを参照して直接ロードすることは避けてください。EXEC ファンクションコールを使うことによって、将来の MS-DOS のバージョンで EXE 形式のファイルのフォーマットが変更されても互換性は保証されます。

割り込みタイプ 27H (プログラムをメモリにとどめたまま終了) の代わりにファンクション 31H (キーププロセス) を使います。ファンクション 31H を使うとプログラムが 64K より大きい場合に問題が生じません。

プログラムの終了にはファンクション 4CH (プロセスの終了) を使います。つぎの手順のいずれかによって終了するプログラムは CS レジスタが PSP (プログラムセグメントプレフィクス) のセグメントアドレスを含んでいることを確認しなくてはなりません。

PSP 内のオフセット 0 にロングジャンプを行う。

INT 20 H を行う (CS:0 は PSP を指していること)

AH=0 で、INT 21 H を行う (CS:0 は PSP を指していること)

AH=0 で、PSP のロケーション 50 H に対するロングコール。

7.7 ファイル・ディレクトリ管理

MS-DOS のファイル管理システムを使います。MS-DOS ファイルシステムを使うことによって、将来の MS-DOS のバージョンに対し、ディスクフォーマットとディスク内のファイル・ディレクトリ管理を通して、プログラムの互換性が保証されます。

FCB の代わりに、ファイルハンドルを使います。ハンドルとは、ファンクション 3CH (ハンドルの作成)、3DH (ハンドルのオープン)、5AH (一時ファイルの作成)、5BH (新しいファイルの作成) によってファイルが解放、または作成される時、MS-DOS が返す 16 ビットの値です。ハンドルを使用する MS-DOS のファイル管理のファンクションリクエストは、1.5.2 の“ファイル管理のファンクションリクエスト”を参照してください。

このコールは、FCB (ファイルコントロールブロック) を使う V2.0 以前のファイル管理のファンクションのかわりに使われます。これはファイル操作が、FCB 情報を操作せず、単にそのハンドルを操作するためです。

FCB を使わなければならない場合は、プログラムが FCB をクローズし、メモリ内に移動さ

せないように気をつけなければなりません。

割り込みタイプ 20H (プログラムの終了), ファンクション 00H (プログラムの終了), ファンクション 4CH (プロセスの終了), ファンクション 0DH (リセットディスク) を実行する前に, 長さを変更したファイルをすべてクローズします。

変更したファイルがクローズされていないと, ファイルの長さが, 正しく書き込まれません。必要のなくなったファイルはすべてクローズします。これによって, ネットワーク環境の状況が最適化されます。

ディスク上のすべてのファイルがクローズされている時だけ, ディスクを変更することができます。内部システムバッファ上の情報が変更を受けた場合, ディスク上に不正確に書き込まれることがあります。

ロックファイル

プログラムはロックされている領域への禁止されているアクセスに依存してはいけません。ロックを試行し, エラーコードを調べることで, 領域の状態を決定します。

プログラムはロックされた領域を含むファイルをクローズしたり, ロックされた領域を含むオープンしたファイルをそのままにして終了することは許されません。この場合, 結果が保証されません。割り込みタイプ 23 H (CTRL-C の抜け出しアドレス), または割り込みタイプ 24 H (致命的エラーによる中断アドレス) によって終了する可能性のあるプログラムは, この割り込みをトラップし, 抜け出す前にすべてのロックされた領域を解放しなければなりません。

7.8 その他のプログラム手順

■ タイミングに対する依存を避けます。

CPU のクロック, 処理速度によってタイミングが異なります。

また, ネットワーク環境内では, タイミングにクロックを使用するプログラムは信頼性が弱くなります。

■ MS-DOS には指定されたインターフェイスを使います。

ハードウェア, またはメディアが変更されても, MS-DOS の提供するインターフェイスを使用していれば, プログラムの変更なしにそれらの機能を使うことができます。

■ 直接, ビデオ RAM をアドレス指定してはいけません。

指定されないファンクションコール, 割り込み, 機能を使ってはいけません。将来の MS-DOS のバージョンで, これらに変更され, (同一名で) 存在するかもしれないからです。これらの機能を使うとプログラムが非常に扱いにくくなります。

■ COM 形式より、EXE 形式を使います。

EXE 形式のファイルはリロケータブル（再配置可能）ですが、COM 形式のファイルはメモリイメージをそのまま持つファイルで、リロケーションのためのコントロール情報が含まれていないためリロケーションは行われません。EXE 形式のファイルは、将来の MS-DOS のバージョンと互換性を保つための拡張可能なヘッダを持っています。

■ 情報をアプリケーションに渡す時は環境を使います。

親プロセスの環境(SET コマンド等で設定された環境変数)は子プロセスに引き継がれます。COMMAND.COM は通常、すべてのアプリケーションの親プロセスになるので、カレントドライブとパス情報を容易にアプリケーションに渡すことができます。

索引

A

ASCIZ150

B

BASIC からのコール20

BIOS パラメータブロック317

BUILD BPB317

C

CONFIG.SYS ファイル305

COMMAND.COM345

COMENT レコード400

CP/M とコンパチブルなコール20

<CTRL-C>検査のセット/リセット(33H) 143

<CTRL-C>の抜け出しアドレス(INT 23H) 30

D

DEINSTALL325

E

EXTDEF レコード389

F

FAT(ファイルアロケーションテーブル) ...349

FAT エントリ350

FCB(ファイルコントロールブロック)16

FCB のフィールド17

FIXUPP レコード394

FLUSH323

G

GRPDEF レコード383

I

INIT312

IOCTL データを得る(4400H)181

IOCTL キャラクタを受け取る(4402H)186

IOCTL キャラクタを送る(4403H)188

IOCTL データをセットする(4401H)184

IOCTL の交換性(4408H)198

IOCTL ビット307

IOCTL ブロックを受け取る(4404H)190

IOCTL ブロックを送る(4405H)192

IOCTL リダイレクトブロック(4409H)200

IOCTL リダイレクトハンドル(440AH) ...202

IOCTL リトライ(440BH)204

L

LEDATA レコード391

LIDATA レコード392

LINNUM レコード390

LNAMES レコード379

M

MEDIA CHECK315

MODEND レコード398

MS-DOS バージョン番号を得る(30H) ...139

MS-Networks13

N

NON FAT ID ビット307

P

PUBDEF レコード387

PSP を得る(62H)278

R	
READ	319
REMOVABLE MEDIA	322
S	
SEGDEF レコード	380
STATUS	323
T	
THEADR レコード	379
TYPDEF レコード	384
U	
USA 規格(国別情報)	150
V	
V2.0 以前のシステムコール	15
W	
WRITE	319
ア	
新しい PSP を作成する(26H)	117
新しいファイルの作成(5BH)	258
アトリビュート(属性)を得る/セットする(43H)	178
アトリビュート(属性)フィールド	307
アブソリュートディスクライト(INT 26H)	38
アブソリュートディスクリード(INT 25H)	36
アロケーションストラテジを得る/セットする(58H)	249
一時ファイルの作成(5AH)	255
一般 IOCTL, ハンドル用(440CH)	206
一般 IOCTL, ブロックデバイス用(440DH)	207
インデックス	369
インテルオブジェクトモジュールフォーマット	363
エラーコード一覧	21
オーバーレイのロード(4B03H)	7, 232
オープン	321
オープンされていない FCB	16
オープンされている FCB	16
カ	
拡張 FCB	18
拡張されたエラーコードを得る(59H)	252
カレントディスクを得る(19H)	95
カレントディレクトリを得る(47H)	218
カレントディレクトリの変更(3BH)	159
環境	355
環境エリア	229
キャラクタデバイス	303
キーププロセス(31H)	141
キーボードステータスの検査(0BH)	68
キーボード入力(08H)	62
キーボード入力とエコー(01H)	51
国別情報を得る(38H)	149
国別情報をセットする(38H)	153
クローズ	321
クロックデバイス	329
子プロセスからリターンコードを得る(4DH)	237
コマンドコードフィールド	310
コマンドプロセッサ	345
コントロールブロック	353
サ	
再試行(リトライ)	35
最初に一致するファイル名の検索(4EH)	238
最初のエントリを検索(11H)	80
シーケンシャルディスクアクセス	75
シーケンシャルライト(15H)	89
シーケンシャルリード(14H)	87
時刻を得る(2CH)	131
時刻をセットする(2DH)	133
システムコール	19
出力ステータスのチェック(4407H)	196
終了アドレス(INT 22H)	30
常駐部	345
初期化部	345
シンボル定義	368
スタック	34
ステータスフィールド	311

ストラテジ	308
ストリングのスクリーン出力(09H)	64
スマート(ブロックデバイス).....	314
セグメントアドレッシング	368
セグメント定義	367
相対レコードのセット(24H)	112

タ

タイプアヘッドバッファ	68, 70
ダム(ブロックデバイス).....	314
直接コンソール I/O(06H)	58
直接コンソール入力(07H)	60
致命的エラーによる中断アドレス(INT 24H) 31	
つぎに一致するファイル名の検索(4FH) ...	240
つぎのエントリを検索(12H)	83
つぎのデバイスヘッダフィールドに対するポイン タ	307
ディスクアロケーション	346
ディスクディレクトリ	346
ディスク転送アドレスを得る(2FH)	137
ディスク転送アドレスのセット(1AH)	97
ディレクトリエントリ	12
ディレクトリエントリの削除(41H)	173
ディレクトリエントリの変更(56H)	244
ディレクトリ管理のファンクションリクエスト	11
ディレクトリの削除(3AH)	157
ディレクトリの作成(39H)	155
ディスクの選択(0EH)	73
ディスクのフリースペースを得る(36H)	147
デバイス管理	409
デバイス管理のファンクションリクエスト ...	10
デバイスストラテジルーチン	305
デバイスドライバ	301
デバイスドライバの登録	305
デバイスドライバの作成方法	304
デバイスドライバファンクション	312
デバイス割り込みルーチン	305
デバイスヘッダ	306
デフォルトドライブのデータを得る(1BH) ...	99
ドライブのデータを得る(1CH)	101
ドライブ名	303

ナ

内部スタック	20
名前フィールド	309
日本規格(国別情報).....	150
入力ステータスのチェック(4406H)	194
抜け出しアドレス	49

ハ

バッファを空にしてキーボード入力(0CH) ...	70
バッファードキーボード入力(0AH)	66
ハンドル	8
ハンドルのオープン(3DH)	163
ハンドルのクローズ(3EH)	167
ハンドルの作成(3CH)	161
非常駐部	345
標準キャラクタデバイス I/O	2
日付を得る(2AH)	127
日付をセットする(2BH)	129
ファイルアクセスのロック(5C00H)	260
ファイルアクセスのロック解除(5C01H) ...	264
ファイルアトリビュート(属性)	12
ファイルアロケーションテーブル(FAT) ...	349
ファイル管理のファンクションリクエスト ...	8
ファイルコントロールブロック(FCB)	16
ファイルシェアリング	9, 163
ファイル・ディレクトリ管理	8, 410
ファイルの大きさを得る(23H)	109
ファイルのオープン(0FH)	75
ファイルのクローズ(10H)	78
ファイルの削除(13H)	85
ファイルの作成(16H)	91
ファイルの日付/時刻を得る/セットする(57H)	246
ファイルハンドルの強制二重化(46H)	216
ファイルハンドルの二重化(45H)	214
ファイルポインタの移動(42H)	175
ファイル名の解析(29H)	124
ファイル名の変更(17H)	93
ファイル名分離記号	125
ファンクションリクエスト(INT 21H)	29
ファンクションリクエスト	43, 408

五十音順・一覧	46
番号順・一覧	43
ブートストラップ	345
プリンタセットアップ(5E02H)	268
ブロックデバイス	303
ブロックデバイスユニット数	303
プログラムをメモリにとどめたまま終了(INT 27H)	41
プログラムセグメント	49, 354
プログラムの終了(INT 20H)	27
プログラムの終了(00H)	49
プログラムのロードと実行(4B00H)	6, 228
プロセス管理	5, 410
プロセスの終了(4CH)	235
ペリファイの状態を得る(54H)	242
ペリファイフラグのセット/リセット(2EH)	135
補助出力(04H)	54
補助入力(03H)	53

マ

マシン名を得る(5E00H)	266
メディアディスクリプタバイト	326
メディアディスクリプタテーブル	326
メモリ管理	3, 409
メモリの割り当て(48H)	220
メモリマップ	353
文字のスクリーン出力(02H)	52
文字のプリンタ出力(05H)	56

ヤ

ユニット	303
ユニットコード	310
ヨーロッパ規格(国別情報)	150

ラ

ライト	319
ライトハンドル(40H)	171
ランダムディスクアクセス	76
ランダムブロックリード(27H)	118
ランダムブロックライト(28H)	121
ランダムライト(22H)	106
ランダムリード(21H)	104

リクエストヘッダ	309
リセット ディスク(0DH)	72
リード	319
リードハンドル(3FH)	169
リロケーションとコントロール情報	359
レコード長	309
レジスタの処理	20
連続非破壊読み込み	320
ロードモジュール	359
論理ドライブマップ(440EH, 440FH)	213

ワ

割り当てられたメモリの解放(49H)	223
割り当てられたメモリブロックの変更(4AH)	226
割り当てリストのエントリを得る(5F02H)	270
割り当てリストのエントリの作成(5F03H)	273
割り当てリストのエントリのキャンセル(5F04H)	276
割り込み・一覧	26
割り込み(タイプ)	26, 407
割り込みベクタを得る(35H)	145
割り込みベクタのセット(25H)	115
ワイルドキャラクタ(カード)	16

NEC

